

AMIGA DEVELOPER CONFERENCE

1990

TABLE OF CONTENTS

Session	Article Title(s)	Section
A3000 Expansion Slot	<i>A3000 Local Bus Expansion Connector</i>	1
A3000 Expansion Bus	<i>The Zorro III Bus Specification</i> <i>A3000 Video Board Form Factor</i>	2
Designing a Zorro III Plug-in Card	<i>BIGRAM</i>	3
Commodore Ethernet/ Arcnet Cards	<i>Arcnet</i>	4
Standard Amiga Network Architecture	<i>SANA: Standard Amiga Network Architecture</i>	5
AmigaVision	<i>AmigaVision -- The Amiga's Multimedia Construction Set</i>	6
CDTV	<i>Commodore Dynamic Totalvision General Specifications</i>	7
CDTV Roundtable	<i>CDTV User Interface Design</i>	8
Publishing and Selling CD-ROM Software	<i>Publishing and Selling CD-ROM Software</i>	9
Preferences	<i>V2.0 Preferences</i>	10
IFF and IFFParse	<i>Using IFF Parse in Applications</i> <i>DPaint ANIM Brush IFF Format</i> <i>ECS Display Modes and ILBM CAMG</i>	11
Debugging Amiga Software	<i>Debugging Tools -- Choosing the Right Tool for the Job</i> <i>Debugging Amiga Software</i>	12
International Marketing	<i>Ten Steps to Better Translations</i>	13
Commodore's OEM/VAR Programs	<i>VAD, VAR, DVAR and OEM Program Summary</i>	14
Scalable Fonts	<i>Scalable Fonts: A Decade of Change</i>	15
Compatibility	<i>Compatibility: How to Make Sure Your Programs Work with V2.0 and Later Versions of the Amiga OS</i>	16
Amiga Standards	<i>Standards</i>	17
The CDTV Development Environment	<i>CTrac Emulation System for CDTV</i> <i>Getting the Best Image for Your CDTV Application</i>	18

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Amiga is a registered trademark of Commodore-Amiga, Inc.

Amiga 500, Amiga 1000, Amiga 2000, AmigaDOS, Amiga Workbench, and Amiga Kickstart are trademarks of Commodore-Amiga, Inc.

Commodore Dynamic Total Vision and CDTV are registered trademarks of Commodore Electronics Limited and Commodore.

AmigaVision is a registered trademark of Commodore Electronics Limited and Commodore.

Amiga/NFS, CBM, Commodore, the Commodore logo, and AUTOCONFIG are registered trademarks of Commodore Electronics Limited.

dBase is a registered trademark of Ashton-Tate Corporation.

68000, 68020, 68030, 68040 and Motorola are trademarks of Motorola, Inc.

Aztec C and Manx are trademarks of Manx Software Systems.

Lattice is a registered trademark of Lattice, Inc.

UNIX is a registered trademark of AT&T.

Novell and Netware are registered trademarks of Novell, Inc.

IBM is a trademark of International Business Machines, Inc.

Macintosh is a trademark licensed to Apple Computer, Inc.
Apple is a trademark of Apple Computer, Inc.

MS-DOS is a trademark of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks.

Amiga is a registered trademark of Commodore-Amiga, Inc.

Amiga 500, Amiga 1000, Amiga 2000, AmigaDOS, Amiga Workbench, and Amiga Kickstart are trademarks of Commodore-Amiga, Inc.

Commodore Dynamic Total Vision and CDTV are registered trademarks of Commodore Electronics Limited and Commodore.

AmigaVision is a registered trademark of Commodore Electronics Limited and Commodore.

Amiga/NFS, CBM, Commodore, the Commodore logo, and AUTOCONFIG are registered trademarks of Commodore Electronics Limited.

dBase is a registered trademark of Ashton-Tate Corporation.

68000, 68020, 68030, 68040 and Motorola are trademarks of Motorola, Inc.

Aztec C and Manx are trademarks of Manx Software Systems.

Lattice is a registered trademark of Lattice, Inc.

UNIX is a registered trademark of AT&T.

Novell and Netware are registered trademarks of Novell, Inc.

IBM is a trademark of International Business Machines, Inc.

Macintosh is a trademark licensed to Apple Computer, Inc.
Apple is a trademark of Apple Computer, Inc.

MS-DOS is a trademark of Microsoft Corporation.

)

)

)



The A3000 Local Bus Expansion Connector

Revision 1.0

By Scott Schaeffer

The local bus connector provides direct access to the A3000 local bus. This allows any card connected to this bus full access to the local bus environment. This bus supports 32 bit burst mode cycles and direct access to the on board 68030. Possible products include CPU accelerators, cache memory boards, high speed bursting RAM expansion, or high speed I/O devices.

Slave Mode Features

The local connector provides direct access to all local bus signals. Any memory or I/O port which resides in this slot must not respond to any address space decoded by either the local bus logic or the Zorro bus. A local bus slot address space has been defined from 0x0800 0000 to 0x0fff ffff. The Gary chip provides a decode of this space on the signal `_RAMSLOT` which is valid at address time. This signal may be ignored and the address decoded by logic on the board if speed is an issue. Local slave devices should also support the signal `_CIIN` if they contain uncachable data.

A signal named `_WAIT` is provided for cache support. Asserting this signal will disable address decoding of Fast RAM on Ramsey and Zorro accesses to Buster. Constraints imposed by the 68030 allow only 18ns to determine a cache hit. It is often more feasible to assert `_STERM` before knowing whether the cycle is a cache hit and rerunning the cycle via `_HALT` and `_BERR` if it is a miss. To achieve this functionality any decoding of the first cycle by Ramsey or Buster must be disabled by asserting `_WAIT` less than 10ns after address valid. If the cycle is determined to be a miss, a rerun is initiated and wait deasserted for the secondary cycle. Assertion of `_WAIT` will keep `_STERM`, `_CBACK`, etc. tristated by Buster or Ramsey and may be controlled by the cache control logic.

Master Mode Features

Bus mastership may be accomplished two ways depending on the desired functionality. The first mode totally disables the motherboard 68030 and its arbitration logic and allows the local bus master to take full arbitration responsibility. The second mode allows the on-board 68030 and the local bus accelerator board to share the bus. This allows multiprocessor capabilities or DMA from the local bus board. This protocol is named fast arbitration and it requires the local bus card and Buster to share the bus request input into the motherboard 68030.

The first mode (arbitration takeover) requires less logic to implement and may be preferred in most implementations. The protocol is as follows. First, the local bus card asserts `_CBR` at power-on to the motherboard which in turn asserts `_BR` to the motherboard 68030. Upon receiving `_BG30` from the motherboard the local card asserts `_BOSS`. Logic on the motherboard uses `_BOSS` to force `_BGACK30` low to the 68030 only, and not the local bus `_BGACK`. In addition the assertion of `_BOSS` tristates `_BG` on the motherboard and in turn the local card should untristate and source its `_BG`. The local card is now the arbitration master and must provide arbitration for the local bus. The onboard 68030 is bus-arbitrated away and never regains the bus.

PAL equations to implement this are as follows:

```
CBR    = 'b'1;    always assert
BOSS   = BG30 # BOSS & !poweron_reset;
BG     = local_card_bg;
BG.oe  = BOSS;
```

All PAL equations are active high and should be inverted in the output stage of the PAL for active low assertion.

`_CBR` is sourced by the local card to the motherboard.

`_BOSS` is sourced by the local card to the motherboard.

`_BG30` is sourced by the motherboard to the local connector.

`local_card_bg` is sourced by the local card in compliance to the operation of arbitration defined in the 68030 users manual.

`poweron_reset` is sourced by the local card and is asserted for a few hundred nanoseconds after poweron. This clears the feedback path on the PAL and may be generated by an RC network which is slewrate cleaned by a Schmitt trigger device.

Timing for takeover mode is not given since all signals are inherently asynchronous. The untristating of `_BG` should be later than the tristating of `_BG` on the motherboard to minimize contention on that signal.

The second mode (fast arbitration) uses the 68030 arbiter to provide cycle arbitration between the local card and the motherboard DMA. Since the 68030 provides only a single bus request input, a scheme referred to as *fast arbitration* is used between the local card and Buster-controlled DMA. Bus request is an open collector line which is time-multiplexed between the local card and Buster. On a positive edge of `CPUCLK`, Buster will assert `_BR` if it requires the bus and `_BR` is not already asserted by the local bus card. On the negative transition of `CPUCLK`, the local bus card may assert `_BR` if it is not already asserted by Buster. This scheme allows both masters to share a single bus request and also requires only 20ns to resolve the master arbitration.

This mode is very efficient but forces the local card to use high-speed logic since the time between clock edges is so short. It is *strongly* suggested that the above logic be incorporated in a 7.5ns or faster registered PAL connected to a F38 open collector device. Clock skew between the clock to this PAL and `CPUCLK` is critical and it is advised that the local card generate and source clocks to the motherboard. Skew between `CPUCLK` and the clock driving this PAL should be less than 2ns.

The PAL equations for `_BR` are as follows:

```
BR_LOCAL.d = !BR & !BR_LOCAL & WANTBUS  
            # BR_LOCAL & !BGACK_LOCAL & !RESET;
```

`BR_LOCAL` is an active high output fed into a 74f38 to produce `_BR`

`_BR` is raw bus request from the local bus connector (keep this trace short on the local card)

`WANTBUS` is the request from the local card which deasserts after it sees `BR_LOCAL` asserted.

`BGACK_LOCAL` is `bgack` asserted by the local card.

`RESET` is any reset signal used to prevent poweron latchup of `BR_LOCAL`.

Note: the above inputs must all meet setup hold requirements of the registered PAL.

After receiving `_BG` from the motherboard 68030, the local card drives `BGACK` (open collector) and assumes mastership of the bus. It may keep the bus for multiple cycles but should not hog the bus for extended periods unless it relinquishes the bus when DMA request is asserted by Buster. Hogging the bus may cause adverse operation of the A3000. Be aware that the local bus signals must be tristated (`AS`, `DSACK`, Address, Data, etc.) prior to deasserting `BGACK`. In addition the local card must not drive the bus or `BGACK` until `AS`, `DSACK`, `BERR`, `HALT`, etc. have deasserted.

It is extremely important that the bus master timing from the local card emulate operation of a 25MHZ or 16MHZ 68030 chip *exactly* as defined by the 68030 user manual. Future enhancements to the A3000 motherboard chips may incorporate rerun cycles and currently incorporate burst cycles and cache coherency signals (`CIIN`). Signals received by the local card from the motherboard provide only the minimum setup and hold required by a 68030. Do not assume for example that data setup from the motherboard will not significantly change, i.e., data setup from Fast RAM on subsequent cycles of a burst is much less than a typical non-burst cycle.

Clock Generation

The A3000 motherboard provides links to disable generation of `CPUCLK` and `CLK90`. This allows the local bus card to maintain better clock skew relationships between its own logic and that of the motherboard. If a local bus card drives the clock lines, the appropriate jumpers must be moved on the motherboard. `CLK90` must be a clock 90 degrees out of phase from `CPUCLK`. It is typically generated from a 5 tap 25ns delay line, where `CLK90` is the 10ns tap when running the system at 25Mhz and `CLK90` is the 15ns tap when at 16Mhz. Note that these clocks are fed through a 74f08 to provide clocking to the motherboard. If the skew generated by the f08 is unacceptable (as in fast arbitration) the socketed f08 may be removed and replaced by a header which shorts the appropriate inputs to outputs. Again careful layout of the clock circuitry is essential for reliable operation.

Local Bus Design Criteria

Any design which plugs into the local bus connector must comply to some basic design rules.

- 1) Due to inductance in the 200 pin connector to vcc and gnd, it is very important to provide ample bypass capacitance in order to maintain good dc vcc and gnd levels.
- 2) All signals from this connector are unbuffered and should not be heavily loaded. A good rule is 2 ttl loads. In addition receivers and drivers should be located near the connector and any connector signal should not run over 4 inches in length from the connector before entering or leaving a driver or receiver.

- 3) Clock generation is especially critical. Keep traces short; ECL routing rules should be followed if possible. Fan out multiple clocks from a single die to minimize loading per clock. Light damping resistors minimize radiation but cause clock distortion, so tune the values carefully.
- 4) Keep in mind current draw on the A3000 is tight so use CMOS and powerdown DRAM modes when possible. New FCT devices use significantly lower current than F and run faster.
- 5) Be aware of heat dissipation issues especially on very high-speed microprocessors.
- 6) Noise test local bus cards and ensure good AC signal quality since a nasty signal will get nastier after passing through an inductive connector to the motherboard.
- 7) Local bus card mounting holes are plated through to ground on the A3000 motherboard and provide an additional low inductance path to ground. Use this path to minimize ground bounce relative to the motherboard.

Local Bus Connector Signal Descriptions

A description of the signal is not given if it is a standard 68030 input or output. Please refer to the 68030 user manual.

<i>Pin number</i>	<i>Signal name</i>	<i>Description</i>
1	<code>_dsack1</code>	
2	<code>gnd</code>	
3	<code>gnd</code>	
4	<code>_halt</code>	
5	<code>r_w</code>	
6	<code>gnd</code>	
7	<code>gnd</code>	
8	<code>_bgack</code>	
9	<code>_sbr</code>	Super DMAC bus request to Buster
10	<code>gnd</code>	
11	<code>gnd</code>	
12	<code>_avec</code>	
13	<code>ext90</code>	clock 90 degrees lagging cpuclock driven by local bus card (optionally)
14	<code>vcc</code>	
15	<code>vcc</code>	
16	<code>_ramslot</code>	Ramsey decode 0x0800 0000 to 0x0fff ffff
17	<code>_boss</code>	takeover arbitration control signal

18	vcc	
19	vcc	
20	fc0	
21	_stern	
22	vcc	
23	vcc	
24	fc1	
25	_br	
26	vcc	
27	vcc	
28	_cback	
29	_berr	
30	reserved	
31	_emul	attached to _CDIS and _MMUDIS on the 68030 and a pullup. Could be used to force disable of MMU and Cache.
32	_cbreq	
33	a8	
34	reserved	
35	gnd	
36	a0	
37	a9	
38	gnd	
39	gnd	
40	a1	
41	a10	
42	reserved	
43	reserved	
44	a2	
45	a11	
46	reserved	
47	gnd	
48	a3	
49	a12	
50	gnd	
51	gnd	
52	a4	
53	a13	
54	reserved	
55	_wait	disable fastram and Zorro decode
56	a5	
57	a14	
58	reserved	
59	gnd	
60	a6	
61	a15	
62	gnd	
63	gnd	
64	a7	
65	a16	
66	reserved	
67	reserved	

68	a24	
69	a17	
70	reserved	
71	gnd	
72	a25	
73	a18	
74	gnd	
75	gnd	
76	a26	
77	a19	
78	reserved	
79	reserved	
80	a27	
81	a20	
82	reserved	
83	gnd	
84	a28	
85	a21	
86	gnd	
87	gnd	
88	a29	
89	a22	
90	reserved	
91	_dsack1	
92	a30	
93	a23	
94	vcc	
95	vcc	
96	a31	
97	_ds	
98	vcc	
99	vcc	
100	_ecs	
101	_ciout	
102	vcc	
103	vcc	
104	_dben	
105	_bg	local bus bg is bg from motherboard 030 tristated by _boss assertion
106	vcc	
107	vcc	
108	_rnc	
109	_cpurst	see A3000 logic for reset definitions
110	_fpurst	
111	reserved	
112	extclk	feeds 74f08 on motherboard which sources cpuc1ka and cpuc1kb. Driven by local bus card (optionally)
113	_ebclr	ebclr is asserted to indicate that a zorro expansion device wants the local bus.
114	reserved	

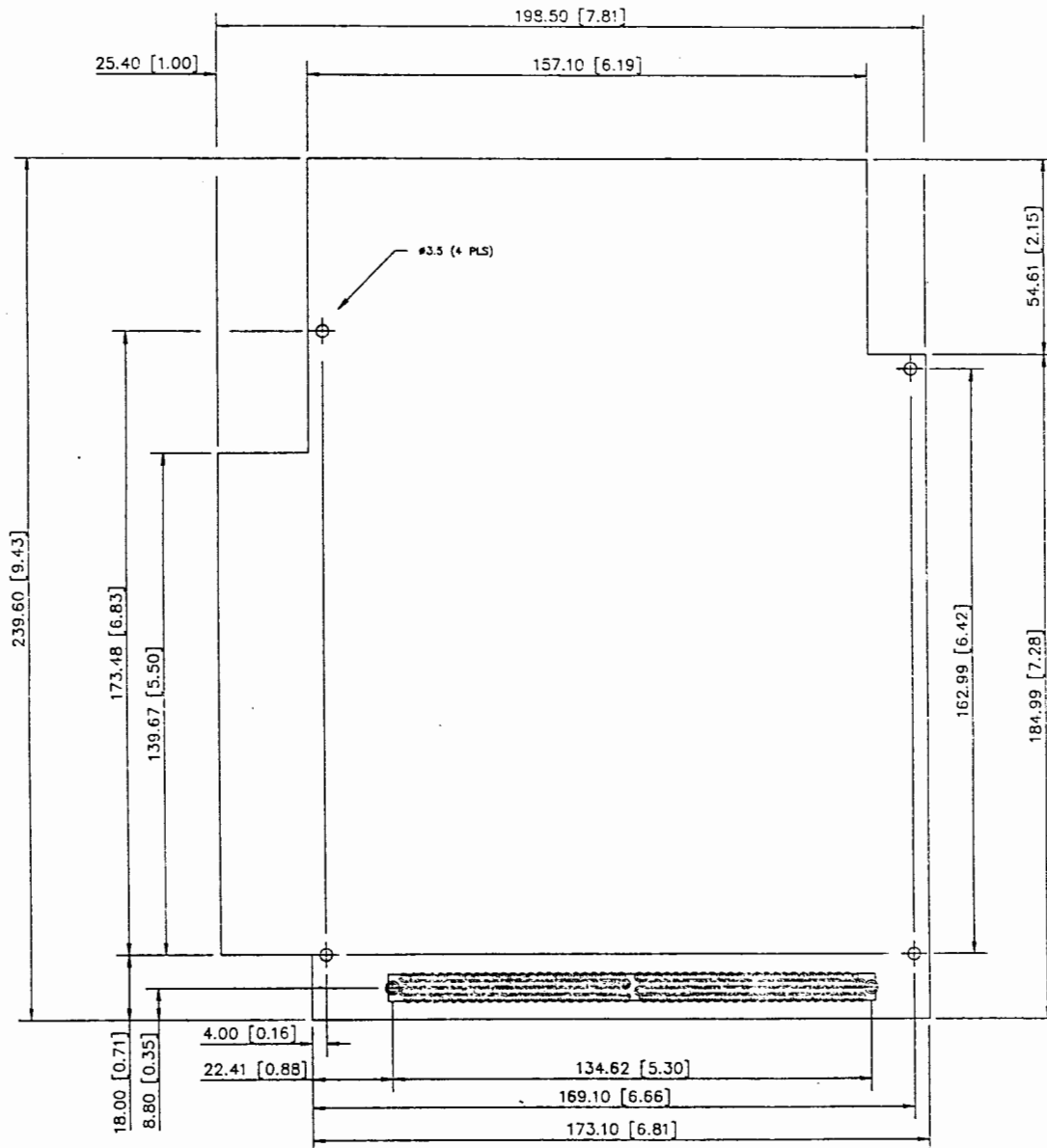
115	gnd	
116	_ipend	
117	_reset	see A3000 logic diagrams for reset definition
118	gnd	
119	gnd	
120	_ipl0	
121	siz0	
122	reserved	
123	reserved	
124	_ipl1	
125	fc2	
126	reserved	
127	clk90	clock which lags cpuc1ka and cpuc1kb by 90 degrees (input only)
128	_ipl2	
129	siz1	
130	gnd	
131	gnd	
132	_ciin	
133	_as	
134	_fpucs	Gary decoded coprocessor select
135	cpuc1ka	one of two motherboard clocks which drive A3000 logic (input only) 25Mhz on 25Mhz system 16Mhz on 16Mhz system.
136	_ocs	
137	d31	
138	gnd	
139	gnd	
140	d15	
141	d30	
142	gnd	
143	gnd	
144	d14	
145	d29	
146	reserved	
147	_cbr	coprocessor bus request, br into motherboard 030 is logical or of _br and _cbr
148	d13	
149	d28	
150	reserved	
151	gnd	
152	d12	
153	d27	
154	gnd	
155	gnd	
156	d11	
157	d26	
158	reserved	

159	_bg30	raw bg from motherboard 68030
160	d10	
161	d25	
162	reserved	
163	gnd	
164	d9	
165	d24	
166	gnd	
167	gnd	
168	d8	
169	d16	
170	reserved	
171	reserved	
172	d0	
173	d17	
174	vcc	
175	vcc	
176	d1	
177	d18	
178	vcc	
179	vcc	
180	d2	
181	d19	
182	vcc	
183	vcc	
184	d3	
185	d20	
186	vcc	
187	vcc	
188	d4	
189	d21	
190	gnd	
191	gnd	
192	d5	
193	d22	
194	gnd	
195	gnd	
196	d6	
197	d23	
198	gnd	
199	gnd	
200	d7	

References

Motorola, MC68030 enhanced 32 bit microprocessor user's manual, second edition,
Motorola Inc. number MC68030UM/AD REV1.





METRIC C.A.D. GENERATED NO MANUAL CHANGES ARE TO BE MADE TO THIS DOCUMENT. © 1990 COMMODORE AMIGA INC. INFORMATION CONTAINED HEREIN IS THE UNPUBLISHED AND CONFIDENTIAL PROPERTY OF COMMODORE AMIGA INC. U.S.G. REPRODUCTION OR DISCLOSURE OF THIS INFORMATION WITHOUT THE PRIOR WRITTEN PERMISSION OF COMMODORE IS STRICTLY PROHIBITED. ALL RIGHTS RESERVED.	UNLESS OTHERWISE SPECIFIED		DRAWN BY: L. HOOPER	DATE: 7-20-90	Commodore 1200 WALTON DRIVE WEST CHESTER, PA. 19380 (610) 431-9100
	TOLERANCES ON DIMENSIONS		DIMS: ENGR:	APPR:	
	UNDER .30 ±0.1 .30 TO .300 ±0.2 OVER .300 ±0.4		USED ON A3000	NEXT ASSY	
	MATERIAL: FINISH:		SCALE 1/1		
				PCB LAYOUT A3000 LOCAL BUS SLOT	
		SIZE C		REV 2	
		SCALE 1/1		SHEET 1 OF 1	

C

C

C



THE ZORRO III BUS SPECIFICATION

**A General Purpose Expansion Bus for
High Performance Amiga Computers**

Document Revision 1.00

Atlanta DevCon Release

by Dave Haynie

June 5, 1990

Copyright © 1990 Commodore Technology

1

2

3

IMPORTANT INFORMATION

"A life spent making mistakes is not only more honorable but more useful than a life spent doing nothing."

-George Bernard Shaw

This Document Contains Preliminary Information

The information contained here, while a honest attempt to get as much Zorro III information down on paper as early and accurately as possible, is still somewhat preliminary in nature and subject to possible errors and omissions. Being early in the life of the Zorro III bus, very few Zorro III cards have yet been designed, so some features described here have not actually been tested in a system, or in some cases, actually implemented as of this writing. That, of course, is one major reason for having a specification in the first place.

Commodore Technology reserves the right to correct any mistake, error, omission, or viscious lie. Corrections will be published as updates to this document, which will be released as necessary in as developer-friendly a manner as possible. Revisions will be tracked via the revision number that appears on the front cover. New revisions will always list the corrections up front, and developers will be kept up to date on released revisions via the normal CATS channels.

All information herein is Copyright © 1990 by Commodore Technology, and may not be reproduced in any form without permission.

ACKNOWLEDGEMENTS

"Art is I; science is we."

-Claude Bernard

I'd like to acknowledge the following people and groups, without whom this new stuff would have been impossible:

- The original Amiga designers, for designing the first microcomputer bus with support for multiple masters, software board configuration, and room to grow.
- The rest of the A3000 Engineers: Greg Berlin, Hedley Davis, Scott Hood, and Scott Schaeffer; PCB master Terry Fisher; and the lab maniacs George Terbush and Brian Fenimore. And of course the boss men, Jeff Porter and Henri Ruben, who let it all happen.
- The folks who helped review this document, overnight: Joe Augenbraun, Dan Baker, Hedley Davis, Bryce Nesbitt, and Jeff Porter.
- The Commodore-Amiga software group, and the Commodore Semiconductor Group, for excellent support in their respective areas.
- Commodore's Developer Support people from both sides of the Atlantic.
- Gold Disk, for some good and relatively bug free electronic publishing software.
- Iggy; an excellent cat, an excellent foot warmer.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1	Intended Audience.....	1-1
1.2	Bug Reports.....	1-2
1.3	Amiga Bus History.....	1-2
1.4	The Zorro III Rationale.....	1-3
1.5	Document Revision History.....	1-4
1.5.1	Changes for Rev 0.90.....	1-5
1.5.2	Changes for Rev 0.91.....	1-5
1.5.3	Changes for Rev 1.00.....	1-5

CHAPTER 2 ZORRO II COMPATIBILITY

2.1	Changes From The A2000 Bus.....	2-2
2.1.1	6800 Bus Interface.....	2-2
2.1.2	Bus Memory Mapping and Cache Support.....	2-2
2.1.3	Bus Synchronization Delays.....	2-3
2.1.4	Zorro II Master Access to Local Slaves.....	2-3
2.1.5	Bus Arbitration and Fairness.....	2-3
2.1.6	Intelligent Cycle Spacing.....	2-3
2.1.7	Bus Drive and Termination.....	2-4
2.1.8	DMA Latency and Overlap.....	2-4
2.1.9	Power Supply Differences.....	2-4
2.2	Bus Architecture.....	2-5
2.3	Signal Description.....	2-5
2.3.1	Power Connections.....	2-6
2.3.2	Clock Signals.....	2-6

2.3.3	System Control Signals.....	2-7
2.3.4	Slot Control Signals.....	2-9
2.3.5	DMA Control Signals.....	2-9
2.3.6	Addressing and Control Signals.....	2-11

CHAPTER 3 BUS ARCHITECTURE

3.1	Basic Zorro III Bus Cycles.....	3-1
3.1.1	Design Goals.....	3-2
3.1.2	Simple Bus Cycle Operation.....	3-2
3.2	Advanced Mode Support Logic.....	3-4
3.2.1	Bus Locking.....	3-4
3.2.2	Cache Support.....	3-5
3.3	Multiple Transfer Cycles.....	3-5
3.4	Quick Bus Arbitration.....	3-7
3.5	Quick Interrupts.....	3-9
3.6	Compatibility with Zorro II Devices.....	3-10

CHAPTER 4 SIGNAL DESCRIPTION

4.1	Power Connections.....	4-1
4.2	Clock Signals.....	4-2
4.3	System Control Signals.....	4-2
4.4	Slot Control Signals.....	4-4
4.5	DMA Control Signals.....	4-4
4.6	Address and Related Control Signals.....	4-5
4.7	Data and Related Control Signals.....	4-7

CHAPTER 5 TIMING

5.1	Standard Read Cycle Timing.....	5-2
5.2	Standard Write Cycle Timing.....	5-4
5.3	Multiple Transfer Cycle Timing.....	5-6
5.4	Quick Interrupt Cycle Timing.....	5-8

CHAPTER 6 ELECTRICAL SPECIFICATIONS

6.1	Expansion Bus Loading.....	6-1
6.1.1	Standard Signals.....	6-2
6.1.2	Clock Signals.....	6-2
6.1.3	Open Collector Signals.....	6-3
6.1.4	Non-bussed Signals.....	6-3
6.2	Slot Power Availability.....	6-3
6.3	Temperature Range.....	6-3

CHAPTER 7	MECHANICAL SPECIFICATIONS	
7.1	Basic Zorro III PIC.....	7-2
7.2	PIC with ISA Option.....	7-3
7.3	PIC with Video Option.....	7-4
 CHAPTER 8	 AUTOCONFIG™	
8.1	The AUTOCONFIG™ Mechanism.....	8-1
8.2	Register Bit Assignments.....	8-2
 APPENDICES		
A.1	Physical and Logical Signal Names.....	A-1
A.2	A Glossary of Terms.....	A-5
A.3	Zorro III Implementations.....	A-9

TABLES AND FIGURES

Figure 1-1	A3000 Memory Map.....	1-4
Figure 2-1	A2000 vs. A3000 Bus Termination.....	2-5
Figure 2-2	Expansion Bus Clocks.....	2-7
Figure 2-3	Zorro II Bus Arbitration.....	2-10
Figure 3-1	Basic Zorro III Cycles.....	3-3
Figure 3-2	Multiple Transfer Read Cycles.....	3-5
Figure 3-3	Zorro III Bus Arbitration.....	3-8
Figure 3-4	Interrupt Vector Cycle.....	3-9
Figure 3-5	Zorro II Within Zorro III.....	3-10
Table 4-1	Memory Space Type Codes.....	4-5
Table 6-1	Zorro III Drive Types.....	6-2
Figure 8-1	Configuration Register Mapping.....	8-2

CHAPTER 1

INTRODUCTION

"Welcome, my son. Welcome to The Machine."

-Pink Floyd

This document describes the complete Zorro III bus, first implemented in the Amiga 3000 Computer. The Zorro III bus is a performance 32 bit expansion bus that is also upward compatible with the Zorro II bus (Amiga 2000 expansion bus). The main intent of the Zorro III bus is to allow fast 32 bit peripherals and memory devices to be added to a high performance Amiga, such as the Amiga 3000, while at the same time allowing standard Zorro II devices to be used wherever they make sense in such a system. This compatibility also insures that the Amiga 3000 will have a number of hardware and software compatible expansion devices available upon introduction, and that Amiga 2000 owners will be able to take their expansion card investment along with them should they migrate to a higher performance Amiga.

1.1 Intended Audience

This document was written primarily for hardware engineers interested in designing Plug In Cards for the Zorro III expansion bus. While it may occasionally be of use to software engineers interfacing to such Zorro III PICs, Amiga system software provides an interface layer (*expansion.library* in the Amiga OS) which manages the needs of most card-level software. A reasonable level of microcomputer knowledge is prerequisite to get much meaning out of these pages. A good understanding of the Motorola 680x0 processors will be quite useful, as will be an understanding of the Zorro II expansion bus used on earlier Amiga computers such as the Amiga 2000.

1.2 Bug Reports

This is the second major publication of the *Zorro III Bus Specification*. While every effort has been made to keep it as accurate as possible, there is certainly the possibility that some errors have made it into this document. Anyone finding any error is encouraged to contact Commodore at the address below:

Dave Haynie/A3000 Systems Engineering
1200 Wilson Drive
West Chester, PA 19380

Bugs can also be reported on BIX or via Usenet; the appropriate BIX conference has yet to be determined, though Dave Haynie can be reached on BIX as "hazy"; for Usenet users, bug reports can be sent to the address "{uunet,rutgers}!cbmvax!bugs"; please also copy any such reports to "{uunet,rutgers}!cbmvax!daveh".

1.3 Amiga Bus History

The original Amiga computer, the Amiga 1000, was introduced in 1985. While it had no built-in standard for expandability, the capability for some form of expansion was considered extremely important; personal computer history up to that date had shown several times that an open hardware expansion capability was often critical to a personal computer's success and to its capability to adapt to new or unusual applications. The A1000 was designed with a connector giving access to the internal 68000 bus and a few other system signals. Shortly after introduction, the formal expansion specification for a card chassis that would connect to the A1000 was published. This bus became commonly known as the Zorro bus*. While the backplane specification was very easy to implement with 1985 PAL technology based on the existing 68000 signals, the specification did incorporate a number of advanced features. Far more sophisticated than the PC-XT and Apple II buses in common use at the time, the Zorro bus allowed any slot to master the bus, and it linked expansion cards with the system software. Addressing jumpers were eliminated, the card's address instead being assigned by software, and cards could easily be identified by software and linked with appropriate driver programs, all with a minimum of user intervention.

With the introduction of the Amiga 2000 system, the Zorro bus was changed slightly. Additional discrete interrupt lines were added, replacing the encoded lines that couldn't easily be used by any bus resident device. As it turns out, these additional encoded lines weren't any more useful, as they couldn't be disabled by software, and as such, they're no longer considered an official part of the Zorro II bus specification (they are supported as part of Zorro III). Finally, the form factor was changed to match that of the IBM PC-AT card, acting as both a cost reduction and allowing the Zorro II bus to offer the PC-AT bus as one optional secondary bus extension. This modified specification became commonly known as the Zorro II bus, and it's the

* The original "Zorro" name comes from the code name of one of the A1000 prototype boards. The "Zorro" board was the one that followed the "Lorraine", and was the board in the works when much of the expansion specifications were worked up. Since everyone uses the "Zorro" name, and no one's suggested a better name, I stick with it throughout this document.

Amiga bus standard that's been in use for most of the Amiga's life. And it's a bus standard that will continue to be important.

1.4 The Zorro III Rationale

With the creation of the Amiga 3000, it became clear that the Zorro II bus would not be adequate to support all of that system's needs. The Zorro II bus would continue to be quite useful, as the current Amiga expansion standard, and so it would have to be supported. A few unused pins on the Zorro II bus and the option of a bus controller custom LSI, gave rise to the Zorro III design, which supports the following features:

- Compatibility with all Zorro II devices.
- Full 32 bit address path for new devices.
- Full 32 bit data path for new devices.
- Bus speed independent of host system CPU speed.
- High speed bus block transfer mode.
- Bus locking for multiprocessor support.
- Cache disable for simple cache support.
- Fair arbitration for all bus masters.
- Cycle by cycle bus arbitration mode.
- High speed interrupt mode.

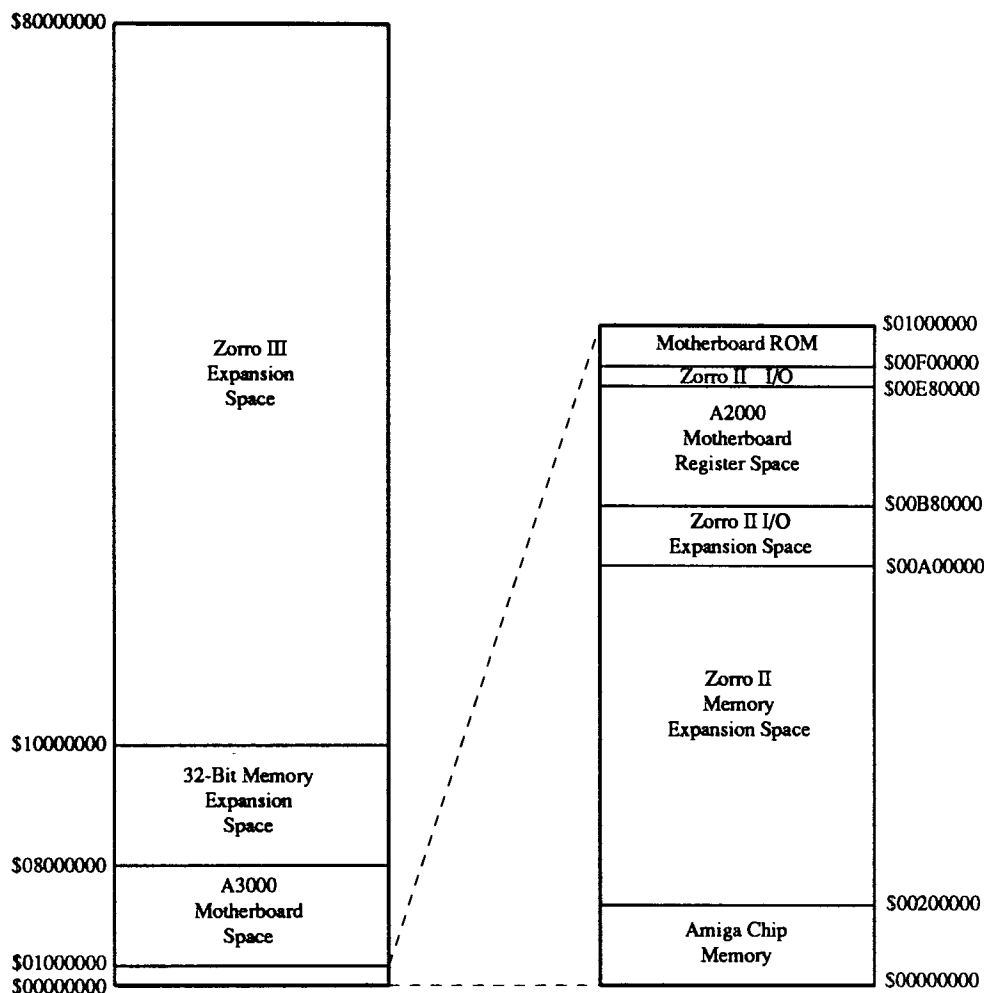
Some of the advanced features, such as burst modes, are designed in such a way as to make them optional; both master and slave arbitrate for them. In addition, it is possible with a bit of extra cleverness, to design a card that automatically configures itself for either Zorro II or Zorro III operation, depending on the status of a sensing pin on the bus.

The Zorro III bus is physically based on the same 100 pin single piece connector as the Zorro II bus. While some bus signals remain unchanged throughout bus operation, other signals change based on the specific bus mode in effect at any time. The bus is geographically mapped into three main sections, *Zorro II Memory Space*, *Zorro II I/O Space*, and *Zorro III Space*. The memory map in *Figure 1-1* shows how these three spaces are mapped in the A3000 system. The Zorro II space is limited to a 16 megabyte region, and since it has DMA access by convention to chip memory, it is in the original 68000 memory map for any bus implementation. The Zorro III space can physically be anywhere in 32 bit memory.

The Zorro III bus functions in one of two different major modes, depending on the memory address on the bus. All bus cycles start with a 32 bit address, since the full 32 bit address is required for proper cycle typing. If the address is determined to be in Zorro II space, a Zorro II compatible cycle is initiated, and all responding slave devices are expected to be Zorro II compatible 16 bit PICs. Should a Zorro III address be detected, the cycle completes when a Zorro III slave responds or the bus times out, as driven by the motherboard logic. It is very important that no Zorro III device respond in Zorro III mode to a Zorro II bus access; as the following chapters will reveal, the two types of cycles make very different use of many of the expansion bus lines, and serious buffer contention can result if the cycle types are somehow

mixed up. The Zorro III bus of course started with the Zorro II bus as its necessary base, but the Zorro III bus mechanisms were designed as much as possible to solve specific needs for high end Amiga systems, rather than extend any particular Zorro II philosophy when that philosophy no longer made any sense. There are actually several variations of the basic Zorro III cycle, though they all work on the same principles. The variations are for optimization of cycle times and for service of interrupt vectors. But all of this in due time.

Figure 1-1: A3000 Memory Map



1.5 Document Revision History

While there's significantly more real Zorro III hardware actually in existence at the time of this writing than when the first revision of this document was created, various Zorro III issues are still, from time to time, changing. In order to document these changes, this section was created. Although revision histories often discuss revisions in reverse chronological order, it's done here in chronological order to keep the subsection numbers consistent between revisions of this document.

1.5.1 Changes for Rev 0.90

The major changes in Rev 0.90 are actually additions. Specifically, the remaining parts of the Zorro III Timing (Chapter 5) and Mechanical (Chapter 7) specifications have been incorporated into this document. Additionally, the Zorro III design example in Appendix A.4 has been deleted. This simple and somewhat kludgy example has been supplanted by a more useful, straightforward, and thoroughly explained example, available as the separate document *BIGRAM 8/32: A Complete Zorro III Design Example*. In general, we expect both documents to be distributed together, but as always, CATS can assist in the procurement of any missing information.

1.5.2 Changes for Rev 0.91

In the Introduction (Chapter 1), the official revision history has been added as a standard part of this document. The Zorro III Bus Architecture (Chapter 3), section 3.5, has been changed to reflect the revised Quick Interrupt vector allocation mechanism. In the Timing specification (Chapter 5), corrections have been made: timing parameter 6 was left out of the section 5.3 timing, and timing parameter 19 was incorrectly specified in section 5.4. In the AUTOCONFIG specification (Chapter 8), corrections have been made to the addressing tables for registers 44 and 48. Also the Quick Interrupt Enable bit (register 08:4) and Vector Register (register 50) have been deleted from the specification. Quick Interrupt Vector allocation is now handled via an Exec call, a single configuration unit can have several vectors, and the means of storage on a PIC is up to the designer.

1.5.3 Changes for Rev 1.00

In the AUTOCONFIG specification (Chapter 8), bit 4 of register 08 has been changed to always read 1 for Zorro III PICs. This change was necessary for compatibility with 1.3, due to a bug in the 1.3 expansion.library. Also, the nybble write configuration mode for the Zorro III configuration block has been eliminated, only byte and word writes are now supported.

CHAPTER 2

ZORRO II COMPATIBILITY

"In Jersey anything's legal, as long as ya don't get caught."

- Traveling Wilburys

The A3000 bus is a rather extensive superset of the A2000 bus design. The compatibility is based on distinct bus modes, rather than a simple extension to the existing bus mechanisms. Through the use of an integrated bus controller (the Fat Buster chip), the expansion bus configures itself differently for the 16 bit A2000-compatible Zorro II modes than the 32 bit Zorro III modes. As a result, while there are still only 100 pins on the expansion bus, some pins change function considerably depending on the bus activity that's currently in progress. While the Zorro II modes of the Zorro III bus are as compatible as possible with the Zorro II bus specification (especially the A2000 implementation of this specification), there are some small differences between the two expansion buses.

Aside from these differences, in general, it's important to understand the Zorro II bus in order to understand the Zorro III bus. The general features of the A3000 bus, like autoconfiguration, the master-slave bus architecture, and the physical attributes come from the Zorro II expansion bus. Other features of the Zorro III bus address shortcomings of the Zorro II architecture, but Zorro II has a hand in how some of these shortcomings are solved under Zorro III. Those with a full understanding of the Zorro II bus will mainly be concerned with the possible bus incompatibilities listed here.

2.1 Changes From The A2000 Bus

While much effort has been made to assure that the Zorro II mode of the A3000 bus is as compatible as possible with the A2000 bus, there are a few points to consider here. Primarily, the A3000's Zorro II modes are driven with a state machine that emulates the 68000 bus protocol. This emulation must be based on the published Motorola specifications detailing 68000 bus behavior. While this has the interesting effect of changing the Zorro II bus from CPU dependent to CPU independent, there's some margin for trouble. Zorro II PICs also designed to these specifications should have no trouble in the A3000 bus in most cases. However, anything designed based on observed 68000 behavior rather than documented 68000 operation is at serious risk of failing in an A3000 bus, as one might expect. There are also actual documented differences, which are listed below.

2.1.1 6800 Bus Interface

A major difference between the A3000 expansion bus in Zorro II mode and the A2000 bus are the absence of the signals /VPA and /VMA, which comprise the 6800/6502 peripheral support mechanism that's part of the 68000 bus interface. This mechanism was never a supported part of the Zorro II specification, however, and it should not be used by any PIC. Any Zorro II PIC that depends on /VPA or /VMA will not work in the A3000 bus. It was, in fact, impossible to legally use this on the A2000 bus. The E clock is, however, supported on the Zorro III bus, though its duty cycle may vary in some situations.

2.1.2 Bus Memory Mapping and Cache Support

Another change to the Zorro II implementation is that the bus mapping logic works a little differently. Zorro II address space is broken up into memory and I/O address space. Memory space is the standard 8 megabyte space from \$00200000-\$009FFFFFFF. The I/O address space is mapped at \$00E80000-\$00FFFFFFF, and a new 1.5 megabyte section (previously reserved for motherboard devices) from \$00A00000-\$00B7FFFF. Zorro II cycles are not generated for non-Zorro II address space, even for 68000 space resources on the local bus. So, for example, a CPU access to chip memory would be visible to a Zorro II PIC in an A2000 backplane, but invisible to that same PIC in an A3000 backplane. Since this extra information on the Zorro II backplane can't be legally used by any PIC anyway, it should not be used by any existing A2000 PICs.

The reason for the two distinct mapping regions is for cache support of Zorro II PICs. All access by the local bus* master to Zorro II memory space results in the local bus cache enable signal being driven and a full port read (eg, both bytes) regardless of the actual data transfer size being requested. A local bus access to Zorro II I/O space results in the local bus cache disable signal being driven and the data strobes for reads indicating the requested transfer size. This cache mapping mechanism was first implemented in the A2630 coprocessor card, so it's not an entirely new concept.

*The *local bus*, motherboard bus, and CPU bus are the same thing; the immediate 680x0 bus connected directly to the CPU in an Amiga computer. Current Amiga computers typically support three distinct buses; the expansion bus, local bus, and chip bus. From the point of view of the expansion bus, the local and chip buses appear as a unified device which may be master or slave to the expansion bus.

2.1.3 Bus Synchronization Delays

Due to the asynchronous nature of the local-to-expansion bus interface for Zorro II cycles, extra wait states may occasionally be added for local to expansion or expansion to local cycles. These are generally manifested as delays between consecutive cycles, since the bus controller is not going to require extra waiting during the cycle -- things will have already been synchronized at that point. The synchronization problems get more difficult for Zorro II master access to local bus slaves, and as a result, wait states here are very common. The actual number of wait states generated in any case will be based on the particular implementation.

2.1.4 Zorro II Master Access to Local Slaves

The only supported local bus resource that's guaranteed accessible to a Zorro II expansion bus master as a slave device is chip bus memory. All I/O devices are implementation dependent and not supportable via DMA. Any attempted access to unsupported local bus resources as expansion slaves will result in an error condition being signalled on both the local and the expansion buses. Most other local bus resources, such as local bus fast memory, are located outside of Zorro II space on most systems and obviously not available to Zorro II masters.

2.1.5 Bus Arbitration and Fairness

The Zorro II bus is now arbitrated fairly. The normal slot-based order of precedence is given to requesting devices, just as in the A2000 implementation. As always, once a bus master assumes bus mastership, it has the bus for as long as it wants the bus (of course, trouble can result if a device takes the bus over for too long). Once a master gives up the bus, it will not be granted it back until all subsequent requests have been serviced. Bus arbitration at its best will be slightly slower than in the A2000 implementation, due to the fairness logic, but it is impossible to jam the arbiter with asynchronous bus requests as in the A2000. The new style arbiter also holds off bus grants while hidden local bus cycles are in progress, so there's no guarantee of a minimum time between bus request and bus grant specified.

2.1.6 Intelligent Cycle Spacing

In order to permit a free intermix of Zorro II and Zorro III cycles, the bus control logic is capable of making intelligent decisions when spacing bus cycles. In some cases, a Zorro II cycle has some component that would naturally extend into a following cycle. The cycle spacing logic detects such a condition, and refuses to start a new cycle until the current one is complete, even if this extends beyond the defined bounds of a Zorro II cycle. For Zorro II PICs that really follow the Zorro II specifications, this should have no effect. However, any Zorro II PIC that holds signals much beyond the end of a cycle, especially critical signals like /SLAVE and /DTACK, will likely incur additional wait states on the Zorro III bus. This is not intended as a license for making sloppy expansion card designs, just an acknowledgement that some Zorro II devices may cause a conflict with the faster Zorro III bus timings, and the best thing to do about such cases is to make them work, even with a possible performance penalty.

2.1.7 Bus Drive and Termination

Finally, the Zorro III bus uses different bus termination than that in the A2000. The Zorro II specification didn't specify the termination expected; backplanes were built that didn't even have termination. The A2000 bus used a circuit consisting of a capacitor in series with a resistor to ground for most of the bus signals. This has good reflection cancelling properties without increasing crosstalk (a major concern on the 2-layer A2000 motherboard), but it does slow things down measurably. The main reason for the change on the A3000 backplane is to support the faster Zorro III bus modes. The multi-layer A3000 motherboard permits a

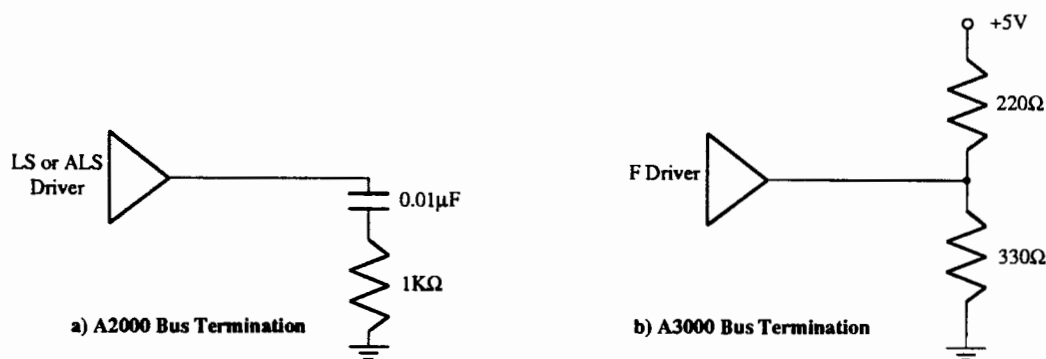


Figure 2-1: A2000 vs. A3000 Bus Termination

reasonably high current bus without undue crosstalk. The thevenin termination makes switching logic levels start from a midpoint instead of a rail, especially for a bus coming out of tri-state (which, based on the Zorro III design, happens constantly). This should not cause problems with Zorro II cards, but it's conceivable that some cards may need to be adjusted to work in this bus (the Zorro III bus requires somewhat higher current capability than the Zorro II bus does. The A3000 does not support enough slots for loading to be a likely problem, but future Zorro III backplanes will have more slots and make this an important consideration).

2.1.8 DMA Latency and Overlap

Zorro II bus masters in a Zorro III backplane will, in many cases, receive a bus grant much sooner than they would in a standard Zorro II backplane. Additionally, in some cases, expansion bus cycles will overlap local bus cycles. The latency incurred on the Zorro II bus during heavy custom chip activity has been greatly reduced for any Zorro III bus master. This should be transparent to the card in question, though it's a good thing to be aware of.

2.1.9 Power Supply Differences

The Zorro II bus is defined as supplying +5VDC @ 2 Amps to each slot, with one slot per backplane supplying 5.0VDC @ 4.0 Amps. The Zorro III bus only provides the 5.0VDC @ 2.0 Amps for each slot.

2.2 Bus Architecture

The Zorro II bus is a simple extension of the 68000 processor bus. Those without a good knowledge of the 68000 local bus will find *The 68000 User's Manual* from Motorola an excellent reference for many Zorro II issues. The *A500/A2000 Technical Reference Manual* from Commodore-Amiga is also required reading for any Zorro II design issues, as it includes a complete description of all the Commodore-Amiga details that aren't part of the 68000 specification.

The basic Zorro II bus is a buffered version of the 68000 processor bus, physically provided on a 100 pin one-piece connector. The bus is 16 bits wide, and provides 24 bits of addressing information. A bus cycle looks exactly like a 68000 bus cycle. The cycle is defined by an address strobe, terminated by a data transfer strobe, and qualified by a read/write strobe, some memory space qualifiers, and one or two byte selection strobes. The basic bus cycle runs for a total of four cycles of a 7.16MHz clock, though it can be extended to add wait states when required.

The Zorro II bus adds a number of features to the basic 68000 CPU bus. It supplies some Amiga system signals that are useful for expansion card designs, such as many of the Amiga system clocks. The bus provides a default data transfer signal, which expansion cards can easily use and modify rather than go to the trouble of creating their own. It provides a number of discrete interrupt lines which are mixed to provide the 68000 with its standard encoded interrupts. The 68000 bus arbitration protocol is used to allow multiple bus masters; arbitration of the bus requests are managed by the Zorro II bus controller to avoid contention between multiple masters. And of course the bus supplies a number of supply voltages for powering cards.

A powerful aspect of the Zorro II bus is its convention for automatically configuring expansion cards, AUTOCONFIG™. On system powerup, the system software interrogates each board to determine what kind of board is installed and how much memory space it needs on the bus. The software then tells each board where to reside in memory. The bus provides hardware lines to allow the boards to be configured in a daisy chained fashion regardless of which slots they occupy and to prevent damage to boards if accidentally configured to reside at the same memory location. Firmware standards also permit software to autoboot or autoinitialize any board, to match soft-loaded device drivers with individual boards, and to link memory boards into the appropriate system memory lists.

2.3 Signal Description

The Zorro II bus can be broken down into various logical signal groups. Some of these groups are unchanged in the Zorro III bus modes, others are drastically different. This section makes note of the original Zorro II name for each signal and the current Zorro III physical pin name for each signal, where different. Some of this information will be repeated in the Zorro III chapters, where appropriate; nothing in this chapter is considered critical to understanding the Zorro III bus, but it is useful. As previously mentioned, the A2000 bus signals unsupported by

the Zorro II specification have been deleted from the Zorro III specification and the A3000 implementation of Zorro III; this section will, however, document those signals for reference purposes. Please see Appendix A for a complete list with pin numbers of the various logical signals that appear on the physical bus during the different phases of the Zorro II and Zorro III bus cycles.

2.3.1 Power Connections

The Zorro III expansion bus provides several different voltages designed to supply expansion devices. There are no changes here that affect Zorro II cards.

Digital Ground (Ground)

This is the digital supply ground used by all expansion cards as the return path for all expansion supplies.

Main Supply (+5VDC)

This is the main power supply for all expansion cards, and it is capable of sourcing large currents; each expansion slot can draw up to 2.0 Amps @ +5VDC. The extra power for one card in any backplane drawing up to 4.0 Amps @ +5VDC is no longer supported.

Negative Supply (-5VDC)

This is a negative version of the main supply, for small current loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 0.3 Amps @ -5VDC for the entire system.

High Voltage Supply (+12VDC)

This is a higher voltage supply, useful for communications cards and other devices requiring greater than digital voltage levels. This is intended for relatively small current loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 8.0 Amps @ +12VDC for the entire system, most of which is normally devoted to floppy and hard disk drive motors, not slots.

Negative High Supply (-12V)

Negative version of the high voltage supply, also commonly used in communications applications, and similarly intended for small loads only. There is no maximum load specified for the Zorro II bus on a per-slot basis; the A2000 implementation specifies 0.3 Amps @ -12VDC for the entire.

2.3.2 Clock Signals

The Zorro III expansion bus provides clock signals for expansion boards. These clocks are for synchronous Zorro II designs and for other synchronous activity such as bus arbitration. While originally based on Amiga local bus clocks, these have no guaranteed relationship to any local bus activity in newer Amiga computers, but are maintained in Amiga computers as part of the expansion bus specification. The relationship between these clocks is illustrated in *Figure 2-2*.

/C1 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the falling edge of the 7M system clock.

/C3 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the rising edge of the 7M system clock.

CDAC Clock

This is a 7.16 MHz system clock (7.09 MHz on PAL systems) which trails the 7M clock by 90° (approximately 35ns).

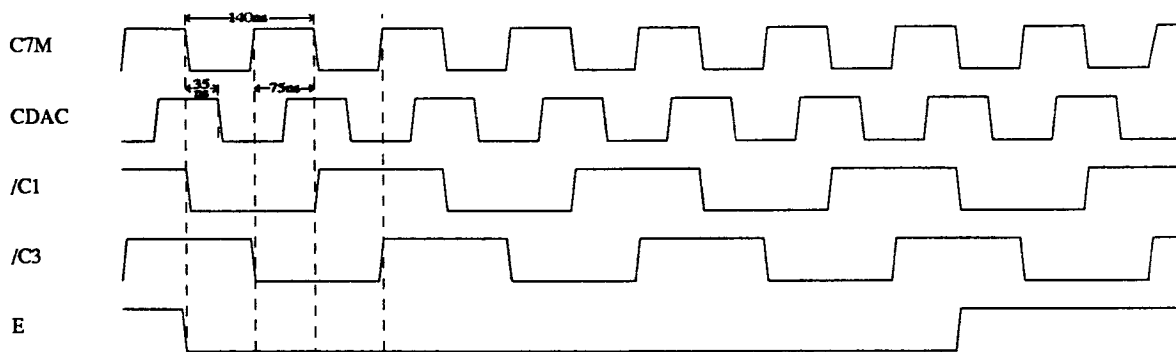


Figure 2-2: Expansion Bus Clocks

E Clock

This is the 68000 generated "E" clock, used for 6800 family peripherals driven by "E" and 6502 peripherals driven by Φ_2 . This clock is four 7M clocks high, six clocks low, as per the 68000 spec. Note that the bus does not support the rest of the 68000's 6800/6502 compatible interface; there may be better ways to clock such devices.

7M Clock

This is the 7.16 MHz system clock (7.09 MHz on PAL systems). This clock forms the basis for all Zorro II/68000 compatible activity, and for various other system functions, such as bus arbitration. .

2.3.3 System Control Signals

The signals in this group are available for various types of system control; most of these have an immediate or near immediate effect on expansion cards and/or the system CPU itself.

Bus Error (/BERR)

This is a general indicator of a bus fault condition. Any expansion card capable of detecting a hardware error relating directly to that card can assert /BERR when that bus error condition is detected, especially any sort of harmful hardware error condition. This

signal is the strongest possible indicator of a bad situation, as it causes all PICs to get off the bus, and will usually generate a level 2 exception on the host CPU. For any condition that can be handled in software and doesn't pose an immediate threat to hardware, notification via a standard processor interrupt is the better choice. The bus controller will drive /BERR in the event of a detected bus collision or DMA error (an attempt by a bus master to access local bus resources it doesn't have valid access permission for). All cards must monitor /BERR and be prepared to tri-state all of their on-bus output buffers whenever this signal is asserted. The current bus master should, if possible, retry the bus cycle after /BERR is negated unless conditions warrant otherwise. Since any number of devices may assert /BERR, and all bus cards must monitor it, any device that drives /BERR must drive with an open collector or similar device capable of sinking at least 12ma, and any device that monitors /BERR should place a minimal load on it (1 "F" type load or less). This signal is pulled high by a passive backplane resistor.

System Reset (/RST, /BUSRST) \equiv (/RESET, /IORST) for Zorro III

The bus supplies two versions of the system reset signal. The /RST signal is bidirectional and unbuffered, allowing an expansion card to hard reset the system. It should only be used by boards that need this reset capability, and is driven only by an open collector or similar device. The /BUSRST signal is a buffered output-only version of the reset signal that should be used as the normal reset input to boards not concerned with resetting the system on their own. All expansion devices are required to reset their autoconfiguration logic when /BUSRST is asserted. This signal is pulled high by a passive backplane resistor.

System Halt (/HLT)

This signal is similar to the 68000 processor halt signal, and is driven by a PIC with an open-collector or similar gate only. Its main use is to indicate a full-system reset. Based on the 68000 conventions, an I/O-only reset, such as initiated by the 680x0 RESET instruction, will drive only /RST and /BUSRST on the bus. A full-system reset, such as a powerup reset or a keyboard reset, drives /HLT low as well. PICs that wish to reset the system CPU as well as the bus and I/O devices drive /RST and /HLT, some bus devices such as processor cards may internally reset only on full-system resets. This signal is pulled high by a passive backplane resistor.

System Interrupts

Six of the decoded 680x0 interrupt inputs are available on the expansion bus, and these are labelled as /INT₂, /INT₆, /EINT₁, /EINT₄, /EINT₅, /EINT₇ on the Zorro II bus. Only the /INT₂ and /INT₆ interrupt inputs are actually supported by Commodore-Amiga as part of the Zorro II specification; the A2000 hardware did not provide the software the required support mechanisms for the safe use of these lines. Each of these interrupt lines are shared by wired ORing, thus each line must be driven by an open-collector or equivalent output type, and all are pulled high by passive backplane resistors.

2.3.4 Slot Control Signals

This group of signals is responsible for the control of things that happen between expansion slots.

Slave (/SLAVE_N)

Each slot has its own /SLAVE output, driven actively, all of which go into the collision detect circuitry. The "N" refers to the expansion slot number of the particular /SLAVE signal. Whenever a Zorro II PIC is responding to an address on the bus, it must assert its /SLAVE output within 35ns of /AS asserted. The /SLAVE output must be negated at the end of a cycle within 50ns of /AS negated. Late /SLAVE assertion on a Zorro II bus can result in loss of data setup times and other problems. A late /SLAVE negation for Zorro II cards can cause a collision to be detected on the following cycle. While the Zorro III sloppy cycle logic eliminates this fatal condition, late /SLAVE negation can nonetheless slow system performance unnecessarily. If more than one /SLAVE output occurs for the same address, or if a PIC asserts its /SLAVE output for an address reserved by the local bus, a collision is registered and results in /BERR being asserted.

Configuration Chain (/CFGIN_N, /CFGOUT_N)

The slot configuration mechanism uses the bus signals /CFGOUT_N and /CFGIN_N, where "N" refers to the expansion slot number. Each slot has its own version of each, which make up the configuration chain between Slots. Each subsequent /CFGIN is a result of all previous /CFGOUTs, going from slot 0 to the last slot on the expansion bus. During the AUTOCONFIG™ process, an unconfigured Zorro PIC responds to the 64K address space starting at \$00E80000 if its /CFGIN signal is asserted. All unconfigured PICs start up with /CFGOUT negated. When configured, or told to "shut up", a PIC will assert its /CFGOUT, which results in the /CFGIN of the next slot being asserted. The backplane passes on the state of the previous /CFGOUT to the next /CFGIN for any slot not occupied by a PIC, so there's no need to sequentially populate the expansion bus slots.

Data Output Enable (DOE)

This signal is used by an expansion card to enable the buffers on the data bus. The main Zorro II use of this line is to keep PICs from driving data on the bus until any other device is completely off the bus and the bus buffers are pointing in the correct direction. This prevents any contention on the data bus.

2.3.5 DMA Control Signals

There are various signals on the expansion bus that coordinate the arbitration of bus masters. Native Zorro III bus masters use some of the same logical signals, but their arbitration protocol is considerably different.

PIC is DMA Owner (/OWN)

This signal is asserted by an expansion bus DMA device when it becomes bus master. This output is to be treated as a wired-OR output between all expansion slots, any of

which may have a PIC signalling bus mastership. Thus, this should be driven with an open-collector or similar output by any PIC using it. This signal is the main basis for data direction calculations between the local and expansion busses, and is pulled up by a backplane resistor.

Slot Specific Bus Arbitration (/BR_N, /BG_N)

These are the slot-specific /BR_N and /BG_N signals, where "N" refers to the expansion slot number. The bus request from each board is taken in by the bus controller and ultimately used to take over the system from 680x0 on the local bus. The bus controller eventually returns one bus grant to the winner among all requesting PICs. From the point of view of

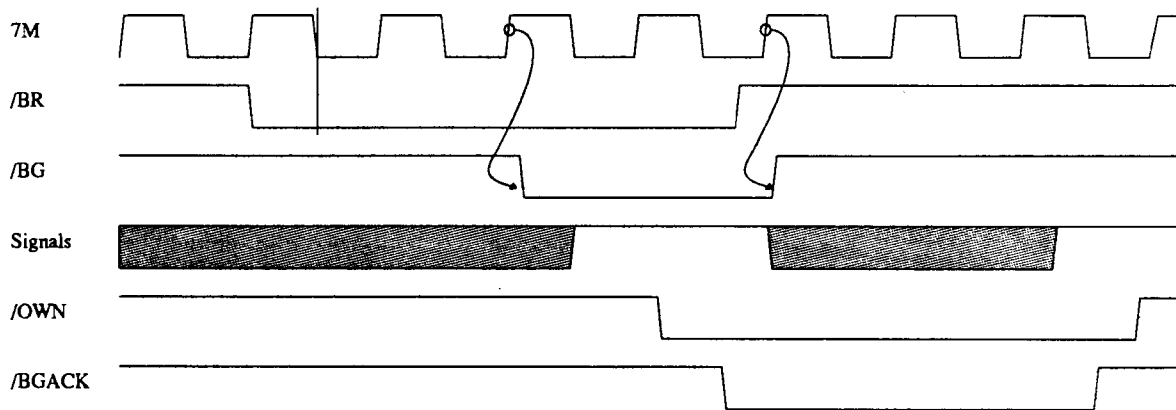


Figure 2-3: Zorro II Bus Arbitration

the individual PIC, the protocol is very similar to that of the 68000 arbitration mechanism. The PIC asserts /BR_N on the rising edge of 7M; some time later, /BG_N is returned on the falling edge of 7M. The PIC waits for all bus activity to finish, asserts /OWN followed by /BGACK, then negates /BR_N, assuming bus mastership. It retains mastership until it negates /BGACK followed by /OWN.

Bus Grant Acknowledge (/BGACK)

Any Zorro II PIC that receives a bus grant asserts this signal as long as it maintains bus mastery. This signal may never be asserted until the bus grant has been received, /AS is negated, /DTACK is negated, and /BGACK itself is negated, indicating that all other potential bus masters have relinquished the bus. This output is driven as a wired-OR output, so all PICs must drive it with an open collector or equivalent device, and a passive pullup is supplied by the backplane.

Bus Want/Clear (/GBG) \equiv (/BCLR) for Zorro III

This signal is asserted by the bus controller to indicate that a PIC wants to master the bus. A bus master assumes that the host CPU wants the bus, and that any time wasted as master is stealing time from the CPU. To avoid such waste, a master should use cache or FIFO to grab slow-coming data, and then transfer it all at once. /BCLR is asserted to indicate that additionally, another PIC wants the bus, and the current bus master should get off as soon as possible. This signal is equivalent to /GBG on the A2000 bus.

2.3.6 Addressing and Control Signals

These signals are various items used for the addressing of devices in Zorro II mode by the local bus and any expansion DMA devices. Most of these signals are very much like 68000 generated bus signals bi-directionally buffered to allow any DMA device on the bus to drive the local bus when such a device is the bus master.

Read Enable (READ)

This is the read enable for the bus, which is equivalent to the 68000's R/W output. READ asserted during a bus cycle indicates a read cycle, READ negated indicates a write cycle. Note that this signal may become valid in a cycle earlier than a 68000 R/W line would, but it remains valid at least as long at the cycle's end.

Address Bus (A₁-A₂₃)

This is logically equivalent to the 68000's address bus, providing 16 megabytes of address space, although much of that space is not assigned to the expansion bus (see the memory map in *Figure 1-1*).

Address Strobe (/AS) ≡ (/CCS) for Zorro III

This is equivalent to the 68000 /AS, called /CCS, for Compatibility Cycle Strobe, in the Zorro III nomenclature. The falling edge of this strobe indicates that addresses are valid, the READ line is valid, and a Zorro II cycle is starting. The rising edge signals the end of a Zorro II bus cycle, signaling the current slave to negate all slave-driven signals as quickly as possible. Note that /CCS, like /AS, can stay asserted during a read-modify-write access over multiple cycle boundaries. To correctly support such cycles, a device must consider both the state of /CCS and the state of the data strobes. Many current Zorro II cards don't correctly support this 680x0 style bus lock.

Data Bus (D₀-D₁₅)

This is a buffered version of the 680x0 data bus, providing 16 bits of data accessible by word or either byte. A PIC uses the DOE signal to determine when the bus is to be driven on reads, and the data strobes to determine when data is valid on writes.

Data Strobes (/UDS, /LDS) ≡ (/DS₃, /DS₂) for Zorro III

These strobes fall on data valid during writes, and indicate byte select for both reads and writes. The lower strobe is used for the lower byte (even byte), the upper strobe is used for the upper byte (odd byte). There is one slight difference between these lines and the 68000 data strobes. On reads of Zorro II memory space, both /DS₃ and /DS₂ will be asserted, no matter what the actual size of the requested transfer is. This is required to support caching of the Zorro II memory space. For Zorro II I/O space, these strobes indicate the actual, requested byte enables, just as would a 68000 bus master.

Data Transfer Acknowledge (/DTACK)

This signal is used to normally terminate both Zorro bus cycles. For Zorro II modes, it is equivalent to the 68000's Data Transfer Acknowledge input. It can be asserted by the

bus slave during a Zorro II cycle at any time, but won't be sampled by the bus master until the falling edge of the S4 state on the bus. Data will subsequently be latched on the S6 falling edge after this, and the cycle terminated with /AS negated during S7. If a Zorro II slave does nothing, this /DTACK will be driven by the bus controller with no wait states, making the bus essentially a 4 cycle synchronous bus. Any slow device on the bus that needs wait states has two options. It can modify the automatic /DTACK negating XRDY to hold off /DTACK. Alternately, it may assert /OVR to inhibit the bus controller's generation of /DTACK, allowing the slave to create its own /DTACK. Any /DTACK supplied by a slave must be driven with an open-collector or similar type output; the backplane provides a passive pullup.

Processor Status (FC0-FC2)

These signals are the cycle type or memory space bits, equivalent for the most part with the 68000 Processor Status outputs. They function mainly as extensions to the bus address, indicating which type of access is taking place. For Zorro II devices, any use of these lines must be gated with /BGACK, since they are not driven valid by Zorro II bus masters. However, when operating on the Zorro III backplane, Zorro II masters that don't drive the function codes will be seen generating an FC1 = 0, which results in a valid memory access. Zorro II cycles are not generated for invalid memory spaces when the CPU is the bus master.

/DTACK Override (/OVR)

This signal is driven by a Zorro II slave to allow that slave to prevent the bus controller's /DTACK generation. This allows the slave to generate its own /DTACK. The previous use of this line to disable motherboard memory mapping, which was unsupported on the A2000 expansion bus, has now been completely removed. The use of XDRY or /OVR in combination with /DTACK is completely up to the board designer -- both methods are equally valid ways for a slave to delay /DTACK. In Zorro III mode, this pin is used for something completely different.

External Ready (XRDY)

This active high signal allows a slave to delay the bus controller's assertion of /DTACK, in order to add wait states. XRDY must be negated within 60ns of the bus master's assertion of /AS, and it will remain negated until the slave wants /DTACK. The /DTACK signal will be asserted by the bus controller shortly following the assertion of XRDY, providing the bus cycle is a S4 or later. XRDY is a wired-OR from all PICs, and as such, must be driven by an open collector or equivalent output. In Zorro III mode, this pin is used for something completely different.

CHAPTER 3

BUS ARCHITECTURE

"We follow in the steps of our ancestry, and that cannot be broken."

-Midnight Oil

While the Zorro II bus design was based in a large part on an already existing bus cycle, the 68000 cycle, the Zorro III bus design had a much different set of preconditions. It is not modeled after any particular CPU specific bus protocol, but instead it's a logical outgrowth of both the need to support Zorro II cards on the same bus and the need to achieve various modern feature and performance goals. These goals were summarized in Chapter 1, now they'll be covered in greater detail here.

3.1 Basic Zorro III Bus Cycles

The basic Zorro III bus cycle is a multiplexed address/data cycle which supplies a full 32 bits worth of address and data per simple cycle. The cycle is a fully asynchronous cycle. The bus master for a given cycle supplies strobes to indicate when address is valid, write data is valid, and read data may be driven. In return, the bus slave for a cycle supplies a strobe to indicate that it is responding to a bus address, and a strobe to indicate that it is done with the bus data for a write cycle, or has supplied valid bus data for a read cycle. The minimum theoretical bus speed is governed only by setup and hold time requirements for the various bus signals. Actual bus speeds is always a function of the bus master and bus slave active for a given cycle. This is considerably different than the way things work under the Zorro II bus, and for several good reasons, which are explained below.

3.1.1 Design Goals

For any computer bus, there are two basic possibilities concerning the fundamental operation of the bus; it's either synchronous or asynchronous. The difference is simple -- the synchronous bus is ultimately tied to a clock of some sort, while the asynchronous bus has no defined relationship to any clock signal. While Motorola specifies the 68000 bus cycle as an asynchronous cycle, they're really referring to the fact that most 68000 inputs are internally synchronized with the bus clock, and therefore, synchronous setup times on the bus do not have to be met to avoid metastability. But the 68000 bus, and the Zorro II bus by extension, are synchronous buses, based on a single bus clock (called E7M on the Zorro II bus). Most Zorro II signals are asserted relative to an edge of the bus clock, and most Zorro II inputs are sampled on an edge of the bus clock. The minimum Zorro II cycle is four bus clocks long, and every wait state added, regardless of the method, will result in a single additional bus clock wait, regardless of the asynchronous appearance of the termination and wait signals on the Zorro II bus.

The Zorro III bus is a fully asynchronous bus, in that all bus events are driven by strobes, and there is no reference clock. The choice of an asynchronous versus a synchronous bus design is governed by the intended application of the bus. Synchronous designs are preferred when a CPU and a memory system (eg, master and slave) can be very tightly coupled to each other. Such designs generally require a tight adherence to timing based on the specific CPU. This is optimal for tightly coupled systems, such the fast memory on the A3000 local bus. Synchronous designs can also be easier to do accurately, as the designer can use clock edges for scheduling events, and there's never any need to waste time in synchronizers to achieve a reliable design.

The design goals for an expansion bus are considerably different. While a fast memory circuit on a system motherboard can change for every new and better design, it's not feasible to require redesign of any significant number of expansion cards every time an improved motherboard design is created. And while a synchronous transfer can be optimal for matched clocks, it can be very inefficient for mismatched CPU and expansion clocks, as synchronizer delays must be introduced for any reliable operation. The A3000 project started with the need to support CPU systems at 16MHz and at 25MHz, and it's obvious that the growth of CPU clock speed will be here for some time to come. Zorro III cards are based on asynchronous handshaking between master and slave in both directions. This means that, as long as masters and slaves manage their own needs, any slave can work with any master. But as masters and slaves improve with technology, bus transfer speeds can automatically increase, without rendering any slower cards obsolete. The Zorro III bus attempts to address the needs of device expansion as much as the needs of memory expansion.

3.1.2 Simple Bus Cycle Operation

The normal Zorro III bus cycle is quite different than the Zorro II bus in many respects. *Figure 3-1* shows the basic cycle. There is no bus clock visible on the expansion bus; the standard Zorro II clocks are still active during Zorro III cycles, but they have no relationship to the Zorro II bus cycle. Every bus event is based on a relationship to a particular bus strobe, and strobes are alternately supplied by master and slave.

A Zorro III cycle begins when the bus master simultaneously drives addressing information on the address bus and memory space codes on the FC_N lines, quickly following that with the assertion of the Full Cycle Strobe, /FCS; this is called the *address phase* of the bus. Any active slaves will latch the bus address on the falling edge of /FCS, and the bus master will tri-state the addressing information very shortly after /FCS is asserted. It's necessary only to latch A₃₁-A₈; the low order A₇-A₂ addresses and FC_N codes are non-multiplexed.

As quickly as possible after /FCS is asserted, a slave device will respond to the bus address by asserting its /SLAVE_N line, and possibly other special-purpose signals. The autoconfiguration process assigns a unique address range to each PIC base on its needs, just as on the Zorro II bus. Only one slave may respond to any given bus address; the bus controller will generate a /BERR signal if more than one slave responds to an address, or if a single slave

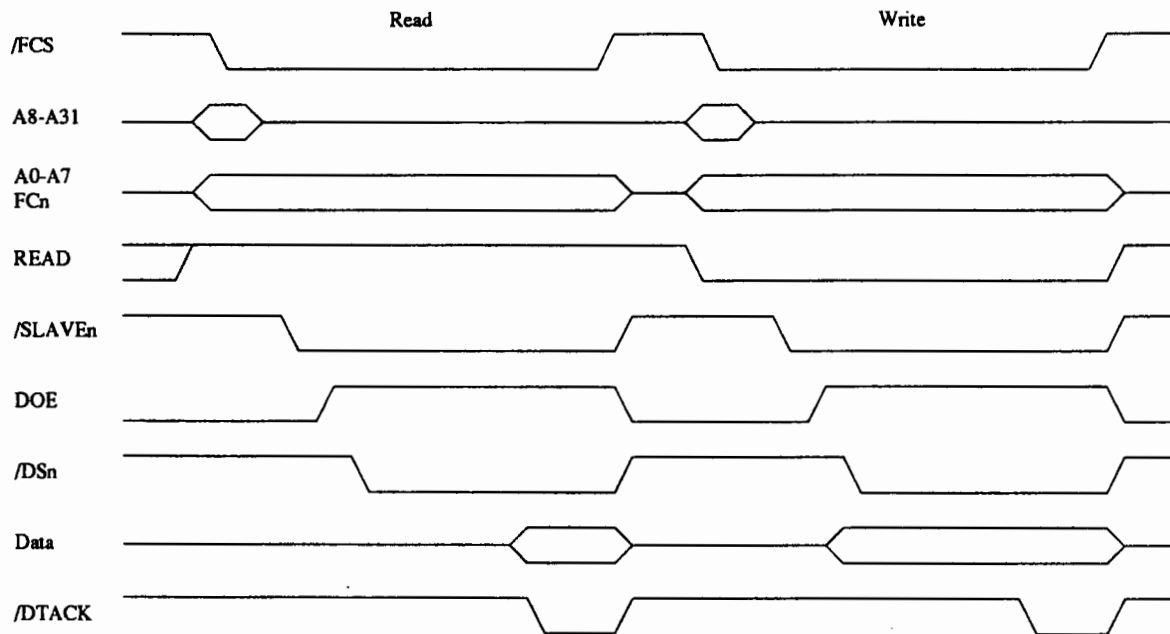


Figure 3-1: Basic Zorro III Cycles

responds to an address reserved for the local bus (this is called a bus collision, and should never happen in normal operation). Slaves don't usually respond to CPU memory space or other reserved memory space types, as indicated by the memory space code on the FC_N lines (see Chapter 4 for details)!

The *data phase* is the next part of the cycle, and it's started when the bus master asserts DOE onto the bus, indicating that data operations can be started. The strobes are the same for both read and write cycles, but of course the data transfer direction is different.

For a read cycle, the bus master drives at least one of the data strobes /DS_N, indicating the physical transfer size requested (however, cachable slaves must always supply all 32 bits of data). The slave responds by driving data onto the bus, and then asserting /DTACK. The bus master then terminates the cycle by negating /FCS, at which point the slave will negate its

/SLAVEN line and tri-state its data. The cycle is done at this point. There are a few actions that modify a cycle termination, those will be covered in later sections.

The write cycle starts out the same way, up until DOE is asserted. At this point, it's the master that must drive data onto the bus, and then assert at least one /DSN line to indicate to the slave that data is valid and which data bytes are being written. The slave has the data for its use until it terminates the cycle by asserting /DTACK, at which point the master can negate /FCS and tri-state its data at any point. For maximum bus bandwidth, the slave can latch data on the falling edge of the logically ORed data strobes; the bus master doesn't sample /DTACK until after the data strobes are asserted, so a slave can actually assert /DTACK any time after /FCS.

3.2 Advanced Mode Support Logic

The Zorro III bus provides support for some more advanced things that weren't generally handled correctly on the Zorro II bus. Amiga computers have traditionally been supporting things that the more mainstream personal computers haven't. High speed DMA transfers and expansion coprocessors such as the Bridge Cards have been with the Amiga since the early days, and high performance main system CPUs with cache memory are now becoming common. The Zorro II bus never properly or easily supported such devices; the Zorro III bus attempts to make support of cache and coprocessor both possible and relatively straightforward. Other new features are covered in later sections.

3.2.1 Bus Locking

The first advanced modification of the basic bus cycle is bus locking, via the /LOCK signal. Bus locking is a hardware convention that allows a bus master to guarantee several cycles will be atomic on the bus. This is necessary to support the sharing of special "mail-box" memory between a bus master and an alternate PIC-based processor; Bridge Cards are an example of this kind of device. The Zorro II bus itself supports bus locking via the 68000 convention. However, the 68000 style of bus locking is often difficult to implement, and support for it was often ignored in Zorro II designs, especially those not directly concerned with multiprocessor support.

The Zorro III mechanism involves no change to the basic bus cycle, other than the monitoring of this /LOCK signal, and as such is much more reasonable to support. The /LOCK signal is asserted by a bus master at address time and maintained across cycles to lock out shared memory coprocessors, allowing hardware backed semaphores to easily be used between such coprocessors. We expect multiprocessing will be a greater concern on the Zorro III bus than it is at present; video coprocessors, RISC devices, and special purpose processors for image processing or mathematics should find a comfortable home on the Zorro III bus.

3.2.2 Cache Support

The other advanced cycle modifier on the Zorro III bus is the cache inhibit line, /CINH. On the Zorro II bus, there was originally no caching envisioned, and therefore no real support for

caching of Zorro II PICs. First in the A2630 and later in the Zorro III bus's emulation of Zorro II, conventions were adopted to permit caching of Zorro II cards. These conventions aren't perfect; MMU tables will sometimes have to supplant this geographic mapping. While Zorro III doesn't have any cache consistency mechanisms for managing caches between several caching bus masters, it does allow cards that absolutely must not be cached to assert a cache inhibit line, /CINH, on a per-cycle basis (asserted at slave time by a responding slave). This cache management is basically the lowest level of a cache management system, mainly useful for support of I/O and other devices that shouldn't be cached. Software will be required for the higher levels of cache management.

3.3 Multiple Transfer Cycles (not supported by Level 1 Fat Buster)

The multiplexed address/data design of the Zorro III bus has some definite advantages. It allows Zorro III cards to use the same 100 pin connector as the Zorro II cards, which results in every bus slot being a 32 bit slot, even if there's an alternate connector in-line with any or all of the system slots; current alternate connectors include Amiga Video and PC-AT (now sometimes called ISA, for *Industry Standard Architecture*, now that it's basically beyond the control of IBM) compatible connectors. This design also makes implementation of the bus controller for a system such as the A3000 simpler. And it can result in lower cost for Zorro III PICs in many cases.

The main disadvantage of the multiplexed bus is that the multiplexing can waste time. The address access time is the same for multiplexed and non-multiplexed buses, but because of the multiplexing time, Zorro III PICs must wait until *data time* to assert data, which places a fixed limit on how soon data can be valid. The Zorro III Multiple Transfer Cycle is a special mode designed to allow the bus to approach the speed of a non-multiplexed design. This mode is especially effective for high speed transfers between memory and I/O cards.

As the name implies, the Multiple Transfer Cycle is an extension of the basic full cycle that results in multiple 32 bit transfers. It starts with a normal full cycle *address phase* transaction, where the bus master drives the 32 bit address and asserts the /FCS signal. A master capable of supporting a Multiple Transfer Cycle will also assert /MTCR at the same time as /FCS. The slave latches the address and responds by asserting its /SLAVEN line. If the slave is capable of multiple transfers, it'll also assert /MTACK, indicating to the bus master that it's capable of this extended cycle form. If either /MTCR or /MTACK is negated for a cycle, that cycle will be a basic full cycle.

Assuming the multiple transfer handshake goes through, the multiple cycle continues to look similar to the basic cycle into the data phase. The bus master asserts DOE (possibly with write data) and the appropriate /DSN, then the slave responds with /DTACK (possibly with read data at the same time), just as usual. Following this, however, the cycle's character changes. Instead of terminating the cycle by negating /FCS, /DSN, and DOE, the master negates /DSN and /MTCR, but maintains /FCS and DOE. The slave continues to assert /SLAVEN, and the bus goes into what's called a *short cycle*.

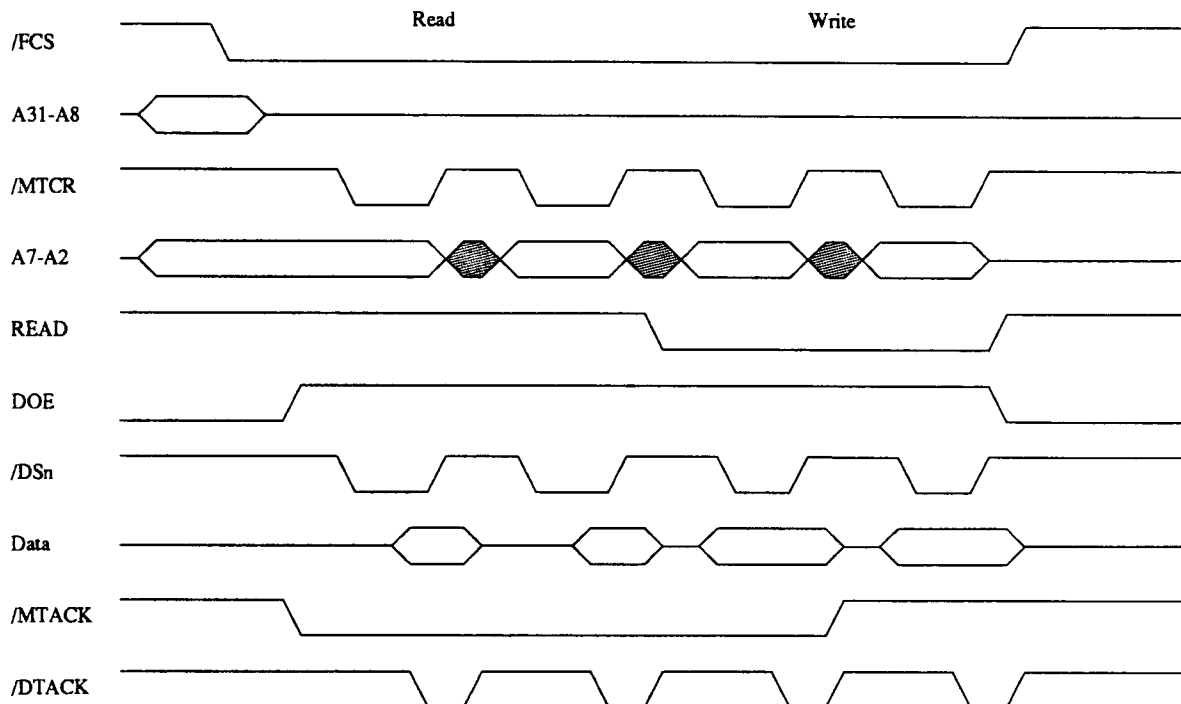


Figure 3-2: Multiple TransferCycles

The short cycle begins with the bus master driving the low order address lines A7-A2; these are the non-multiplexed addresses and can change without a new *address phase* being required (this is essentially a page mode, fully random accesses on this 256 byte page). The READ line may also change at this time. The master will then assert /MTCR to indicate to the slave that the short cycle is starting. For reads, the appropriate /DSn are asserted simultaneously with /MTCR, for writes, data and /DSn are asserted slightly after /MTCR. The slave will supply data for reads, then assert /DTACK, and the bus will terminate the short cycle and start into either another short cycle or a full cycle, depending on the multiple cycle handshaking that has taken place.

The question of whether a subsequent cycle will be a full cycle or a short cycle is answered by multiple cycle arbitration. If the master can't sustain another short cycle, it will negate /FCS and DOE along with /MTCR at the end of the current short cycle, terminating the full cycle as well. The master always samples the state of /MTACK on the falling edge of /MTCR. If a slave can't support additional short cycles, it negates /MTACK one short cycle ahead of time. On the following short cycle, the bus master will see that no more short cycles can be handled by the slave, and fully terminate the multiple transfer cycle once this last short cycle is done.

PICs aren't absolutely required to support Multiple Transfer Cycles, though it is a highly recommended feature, especially for memory boards. And of course, all PICs must act intelligently about such cycles on the bus; a card doesn't request or acknowledge any Multiple Transfer Cycle it can't support.

3.4 Quick Bus Arbitration (not supported by Level 1 Fat Buster)

The Zorro II bus does an adequate job of supporting multiple bus masters, and the Zorro III bus extends this somewhat by introducing fair arbitration to Zorro II cards. However, some desirable features cannot be added directly to the Zorro II arbitration protocol. Specifically, Zorro III bus arbitration is much faster than the Zorro II style, it prohibits bus hogging that's possible under the Zorro II protocol, and it supports intelligent bus load balancing.

Load balancing requires a bit of explanation. A good analogy is to that of software multitasking; there, an operating system attempts to slice up CPU time between all tasks that need such time; here, a bus controller attempts to slice up bus time between all masters that need such time. With preemptive multitasking such as in the Amiga and UNIX OSs, equal CPU time can be granted to every task (possibly modified by priority levels), and such scheduling is completely under control of the OS; no task can hog the CPU time at the expense of all others. An alternate multitasking scheme is a popular add-on to some originally non-multitasking operating systems lately. In this scheme, each task has the CPU until it decides to give up the CPU, basically making the effectiveness of the CPU sharing at the mercy of each task. This is exactly the same situation with masters on the Zorro II bus. The Zorro III arbitration mechanism attempts to make bus scheduling under the control of the bus controller, with masters each being scheduled on a cycle-by-cycle basis.

When a Zorro III PIC wants to master the bus, it *registers* with the bus controller. This tells the bus controller to include that PIC in its scheduling of the expansion bus. There may be any number of other PICs registered with the bus controller at any given time. The CPU is always scheduled expansion bus time, and other local bus devices, such as a hard disk controller, may be registered from time to time.

Once registered, a PIC sits idle until it receives a *grant* from the bus controller. A grant is permission from the bus controller that allows the PIC to master the Zorro III bus for one full cycle. A PIC always gets one full cycle of bus time when given a grant, and assuming it stays registered, it may receive additional full cycles. Within the full cycle, the PIC may run any number of Multiple Transfer Cycles, assuming of course the responding slave supports such cycles. For multiprocessor support, a PIC will be granted multiple atomic full cycles if it locks the bus. This feature is only for support of hardware semaphores and other such multiprocessor needs; it is not intended as a means of bus hogging!

Figure 3-3 shows the basics of Zorro III bus arbitration, which is pretty simple. While it uses some of the same signals as the 680x0 inspired Zorro II bus arbitration mechanism, it has nothing to do with 680x0 bus arbitration; the /BRN and /BGN signals should be thought of as completely new signals. In order to register with the bus controller as a bus master, a PIC asserts its private /BRN strobe on the rising edge of the 7M clock, and negates it on the next rising edge. The bus controller will indicate mastership to a registered bus master by asserting its /BGN. Once granted the bus, the PIC drives only the standard cycle signals: addresses, /FCS, /EDSN, data, etc. in a full cycle. The bus controller manages the assertion of /OWN and /BGACK, which are important only for bus management and Zorro II support. While a scheduling scheme

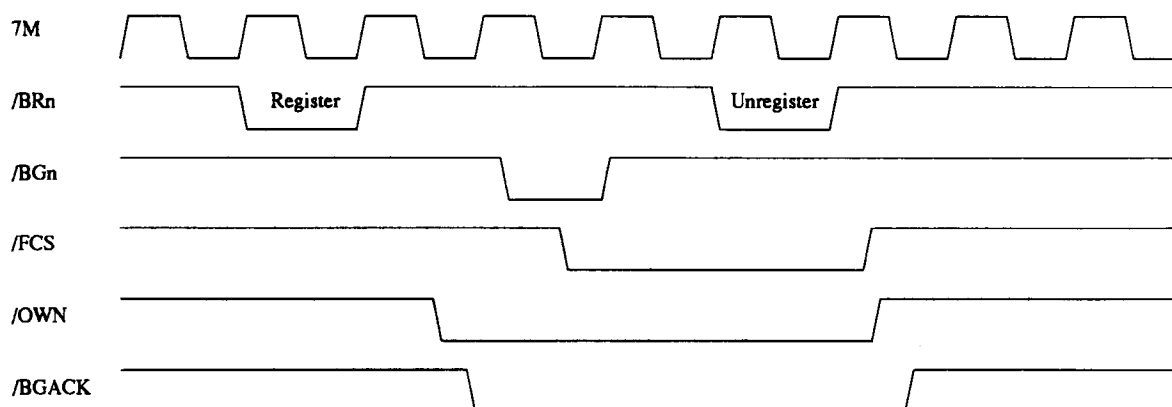


Figure 3-3: Zorro III Bus Arbitration

isn't part of this bus specification, the bus master will only be guaranteed one bus cycle at a time. The **/BGN** line is negated shortly after the master asserts **/FCS** unless the bus controller is planning to grant multiple full cycles to the master. The only thing that'll force the controller to grant multiple full cycles is a locked bus. Any master that works better with multiple cycles, such as devices with buffers to empty into memory, should run a Multiple Transfer Cycle to transfer several longwords during the same full cycle. For this reason, slave cards are encouraged to support Multiple Transfer Cycles, even if they don't necessarily run any faster during them.

Once a registered bus master has no more work to do, it unregisters with the bus controller. This works just like registering -- the PIC asserts **/BRN** on the rise of **7M**, then negates it on next rising **7M**. This is best done during the last cycle the bus master requires on the bus. If a registered master gets a grant before unregistering and has no work to do, it can unregister without asserting **/FCS**, to give back the bus without running a cycle. It's always far better to make sure that the master unregisters as quickly as possible. Bus timeout causes an automatic unregistering of the registered master that was granted that timed-out cycle; this guarantees that an inactive registered master can't drag down the system. If a master sees a **/BERR** during a cycle, it should terminate that cycle immediately and re-try the same cycle. If the retried cycle results in a **/BERR** as well, nothing more can be done in hardware; notification of the driver program is the usual recourse.

The bus controller may have to mix Zorro II style bus arbitration in with Zorro III arbitration, as Zorro II and Zorro III cards can be freely mixed in a backplane. Because of this, Multiple Transfer Cycles, and the self-timed nature of Zorro III cards, there's no way to guarantee the latency between bus grants for a Zorro III card. The bus controller does, however, make sure that all masters are fairly scheduled so that no starvation occurs, if at all possible. Zorro III cards must use Zorro III style bus arbitration; although current Zorro III backplanes can't differentiate between Zorro II and Zorro III cards when they request (other than by the request mechanism), it can't be assumed that a backplane will support Zorro III cycles with Zorro II mastering, or visa-versa.

3.5 Quick Interrupts (not supported by Level 1 Fat Buster)

While the Zorro II bus has always supported shared interrupts, the Zorro III bus supports a mechanism wherein the interrupting PIC can supply its own vector. This has the potential to make such vectored interrupts much faster than conventional Zorro II chained interrupts, arbitrating the interrupting device in hardware instead of software.

A PIC supporting quick interrupts has on-board registers to store one or more vector numbers; the numbers are obtained from the OS by the device driver for the PIC, and the PIC/driver combination must be able to handle the situation in which no additional vectors are available. During system operation, this PIC will interrupt the system in the normal manner, by

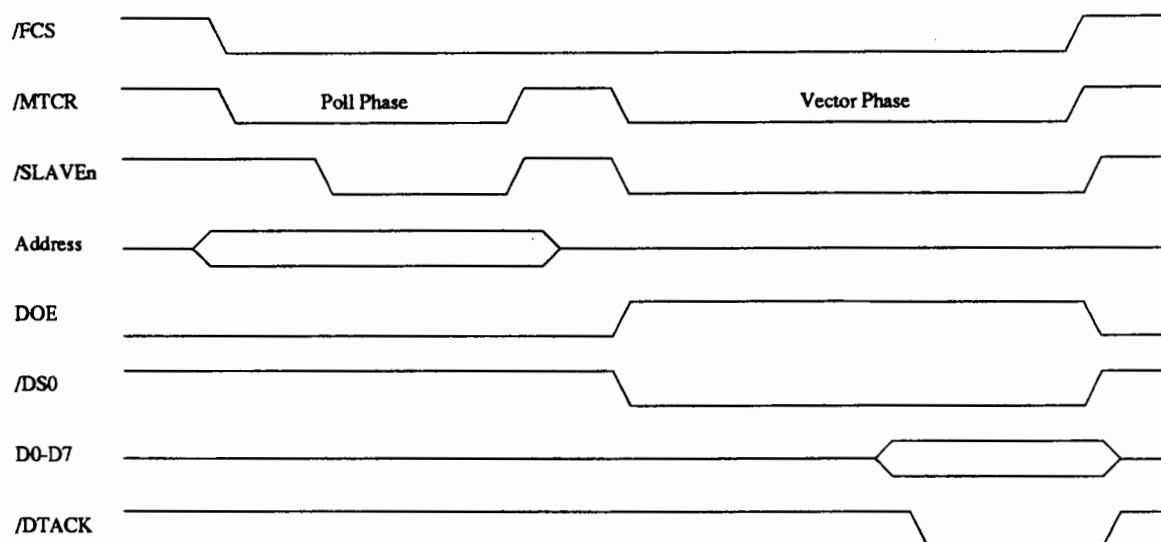


Figure 3-4: Interrupt Vector Cycle

asserting one of the bus interrupt lines. This interrupt will cause an interrupt vector cycle to take place on the bus. This cycle arbitrates in hardware between all PICs asserting that interrupt, and it's a completely different type of Zorro III cycle, as illustrated in Figure 3-4.

The bus controller will start an interrupt vector cycle in response to an interrupt asserted by any PIC. This cycle starts with /FCS and /MTCR asserted, a FC code of 7 (CPU space), a CPU space cycle type, given by address lines A16-A19, of 16, and the interrupt number, which is on A1-A3. At this point, called the *polling phase*, any PIC that has asserted an interrupt and wants to supply a vector will decode the MS lines, the cycle type, match its interrupt number against the one on the bus, and assert /SLAVEn if a match occurs. Shortly thereafter, the /MTCR line is negated, and the slaves all negate /SLAVEn. But the cycle doesn't end.

The next step is called the *vector phase*. The bus controller asserts one /SLAVEn back to one of the interrupting PICs, along with /MTCR and /DS0, but no addresses are supplied. That PIC will then assert its 8 bit vector onto D0-D7 of the 32 bit data bus and /DTACK, as quickly as possible, thus terminating the cycle. The speed here is very critical; an automatic autovector

timeout will occur very quickly, as any actual waiting that's required for the quick interrupt vector is potentially delaying the autovector response for Zorro II style interrupts. A PIC stops driving its interrupt when it gets the response cycle; it must also be possible for this interrupt to be cleared in software (eg, the PIC must make choice of vectoring vs. autovectoring a software issue).

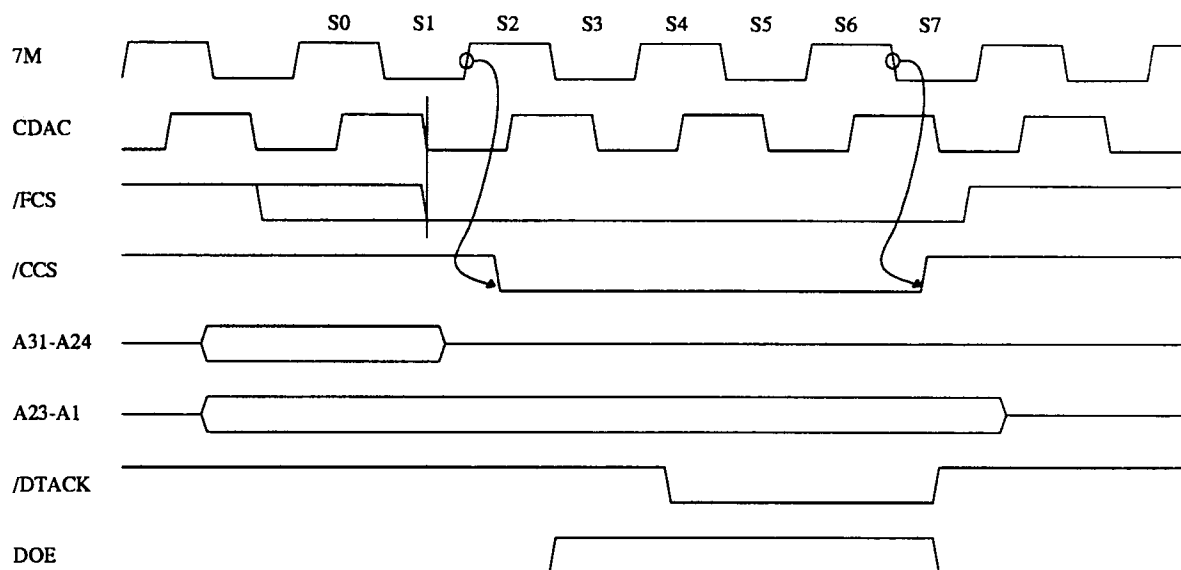


Figure 3-5: Zorro II Within Zorro III

3.6 Compatibility with Zorro II Devices

As detailed in Chapter 2, the Zorro III bus supports a bus cycle mode very similar to the 68000-based Zorro II bus, and is expected to be compatible with all properly designed Zorro II PICs. As shown in *Figure 1-1*, Zorro II and Zorro III expansion spaces are geographically mapped on the Zorro III bus. The mapping logic resides on the bus, and operates on the bus address presented for any cycle. Every cycle starts out assuming a Zorro III cycle, but the mapping logic will inscribe a Zorro II cycle within the Zorro III cycle if the address range is right. *Figure 3-5* details the bus action for this mode.

The cycle starts out with the usual address phase activity; the bus master asserts /FCS after asserting the full 32 bit address onto the address bus. The bus decoder maps the bus address asynchronously and quickly, so that by the time /FCS is asserted, the memory space is determined. A Zorro II space access will cause A8-A23 to remain asserted, rather than being tri-stated along with A24-A31, as the Zorro III cycle normally does. The bus controller synchs the asynchronous /FCS on the falling edge of CDAC, then drives /CCS (the /AS equivalent) out on the rising edge of 7M, based on that synched /FCS. For a read cycle, /DS3 and/or /DS2 (the /UDS and /LDS replacements, respectively) would be asserted along with /CCS; write cycles see those lines asserted on the next rising edge of 7M, at S4 time. The DOE line is also asserted at the start of S4.

The bus controller starts to sample /DTACK on the falling edge of 7M between S₄ and S₅, adding wait states until /DTACK is encountered. As per Zorro II specs, the PIC need not create a /DTACK unless it needs that level of control; there are Zorro II signals to delay the controller-generated /DTACK, or take it over when necessary. The controller will drive its automatic /DTACK at the start of S₄, leaving plenty of time for the sampling to come at S₅. Once a /DTACK is encountered, cycle termination begins. The controller latches data on the falling 7M edge between S₆ and S₇, and also negates /CCS and the /DSN at this time. Shortly thereafter, the controller negates /DTACK (when controlling it), DOE, and tri-states the data bus, getting ready for the next cycle.

CHAPTER 4

SIGNAL DESCRIPTION

*"Pushing back the limits of human achievement, reaching for the stars,
that's not something we do. It's what we are."*

-Michael Swaine

The signals detailed here are the Zorro III mode signals. While some of this information is the same in as the Zorro II signal description of Chapter 2, many like-seeming bus signals behave differently in Zorro III mode than Zorro II mode. These can be a very important differences; thus the complete set of signals is detailed here.

4.1 Power Connections

The expansion bus provides several different voltages designed to supply expansion devices. These are basically the same for the Zorro III bus as they were for the Zorro II bus, with the exception of one pin, and that the specification has been clarified a bit. Note that all Zorro III PICs must list their power consumption specifications.

Digital Ground (Ground)

This is the digital supply ground used by all expansion cards as the return path for all expansion supplies.

Main Supply (+5VDC)

This is the main power supply for all expansion cards, and it is capable of sourcing large currents; each PIC can draw up to 2.0 Amps @ +5VDC.

Negative Supply (-5VDC)

This is a negative version of the main supply, for small current loads only; each PIC can draw up to 60 mA @ -5VDC.

High Voltage Supply (+12VDC)

This is a higher voltage supply, useful for communications cards and other devices requiring greater than digital voltage levels. This is intended for relatively small current loads only; each PIC can draw up to 500mA @ +12VDC.

Negative High Supply (-12V)

Negative version of the high voltage supply, also used in communications applications, and similarly intended for small loads only; each PIC can draw up to 60 mA @ -12VDC.

4.2 Clock Signals

The expansion bus provides clock signals for expansion boards. The main use for these clocks on Zorro III cards is bus arbitration clocking. There is no relationship between any of these clocks and normal Zorro III bus activity. The relationship between these clocks is illustrated in *Figure 2-2*.

/C1 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the falling edge of the 7M system clock.

/C3 Clock

This is a 3.58 MHz clock (3.55 MHz on PAL systems) that's synched to the rising edge of the 7M system clock.

CDAC Clock

This is a 7.16 MHz system clock (7.09 MHz on PAL systems) which trails the 7M clock by 90° (approximately 35ns).

E Clock

This is the 68000 generated "E" clock, used for 6800 family peripherals driven by "E" and 6502 peripherals driven by Φ_2 . This clock is four 7M clocks high, six clocks low, as per the 68000 spec.

7M Clock

This is the 7.16 MHz system clock (7.09 MHz on PAL systems). This clock drives the bus master registration mechanism for Zorro III bus masters.

4.3 System Control Signals

The signals in this group are available for various types of system control; most of these have an immediate or near immediate effect on expansion cards and/or the system CPU itself.

Hardware Bus Error/Interrupt (/BERR)

This is a general indicator of a bus fault condition of some kind. Any expansion card capable of detecting a hardware error relating directly to that card can assert /BERR when that bus error condition is detected, especially any sort of harmful hardware error condition. This signal is the strongest possible indicator of a bad situation, as it causes all PICs to get off the bus, and will usually generate a level 2 exception on the host CPU. For any condition that can be handled in software and doesn't pose an immediate threat to hardware, notification via a standard processor interrupt is the better choice. The bus controller will drive /BERR in the event of a detected bus collision or DMA error (an attempt by a bus master to access local bus resources it doesn't have valid access permission for). All cards must monitor /BERR and be prepared to tri-state all of their on-bus output buffers whenever this signal is asserted. An expansion bus master will attempt to retry a cycle aborted by a single /BERR and notify system software in the case of two subsequent /BERR results. Since any number of devices may assert /BERR, and all bus cards must monitor it, any device that drives /BERR must drive with an open collector or similar device, and any device that monitors /BERR should place a minimal load on it. This signal is pulled high by a passive backplane resistor.

System Reset (/RESET, /IORST)

The bus supplies two versions of the system reset signal. The /RESET signal is bidirectional and unbuffered, allowing an expansion card to hard reset the system. It should only be used by boards that need this reset capability, and is driven only by an open collector or similar device. The /IORST signal is a buffered output-only version of the reset signal that should be used as the normal reset input to boards not concerned with resetting the system on their own. All expansion devices are required to reset their autoconfiguration logic when /IORST is asserted. These signals are pulled high by passive backplane resistors.

System Halt (/HLT)

This signal is driven, along with /RESET, to assert a full-system reset. A full-system reset is asserted on a powerup reset or a keyboard reset; any PIC that needs to differentiate between full system and I/O reset should monitor /HLT and /IORST unless it also needs to drive a reset condition. This is driven with an open-collector output, or the equivalent, and pulled up by a backplane resistor.

System Interrupts

Six of the decoded 680x0 interrupt inputs are available on the expansion bus, and these are labelled as /INT₁, /INT₂, /INT₄, /INT₅, /INT₆, /INT₇. Each of these interrupt lines is shared by wired ORing, thus each line must be driven by an open-collector or equivalent output type. Zorro III interrupts can be handled Zorro II style, via autovectors and daisy-chained polling, or they can be vectored using the quick interrupt protocol described in Chapter 3. Zorro III cards may use any of these interrupt lines; system hardware supports them all properly. Any card that is designed for operation in both Zorro II and Zorro III backplanes must only use /INT₂ or /INT₆. These signals are pulled high by passive backplane resistors.

4.4 Slot Control Signals

This group of signals is responsible for the control of things that happen between expansion slots.

Slave (/SLAVEN)

Each slot has its own /SLAVEN output, driven actively, all of which go into the collision detect circuitry. The "N" refers to the expansion slot number of the particular /SLAVE signal. Whenever a Zorro III PIC is responding to an address on the bus, it must assert its /SLAVEN output very quickly. If more than one /SLAVEN output occurs for the same address, or if a PIC asserts its /SLAVEN output for an address reserved by the local bus, a collision is registered and the bus controller asserts /BERR. The bus controller will assert /SLAVEN back to the interrupting device selected during a Quick Interrupt cycle, so any device supporting Quick Interrupts must be capable of tri-stating its /SLAVEN; all others can drive SLAVEN with a normal active output.

Configuration Chain (/CFGIN_N, /CFGOUT_N)

The slot configuration mechanism uses the bus signals /CFGOUT_N and /CFGIN_N, where "N" refers to the slot number. Each slot has its own version of both signals, which make up the *configuration chain* between slots. Each subsequent /CFGIN_N is a result of all previous /CFGOUTs, going from slot 0 to the last slot on the expansion bus. During the autoconfiguration process, an unconfigured Zorro III PIC responds to the 64K address space starting at either \$00E80000 or \$FF000000 if its /CFGIN_N signal is asserted. All unconfigured PICs start up with /CFGOUT_N negated. When configured, or told to "shut up", a PIC will assert its /CFGOUT_N, which results in the /CFGIN_N of the next slot being asserted. Backplane logic automatically passes on the state of the previous /CFGOUT_N to the next /CFGIN_N for any slot not occupied by a PIC, so there's no need to sequentially populate the expansion bus slots.

Backplane Type Sense (SenseZ3)

This line can be used by the PIC to determine the backplane type. It is grounded on a Zorro II backplane, but floating on a Zorro III backplane. The Zorro III PIC connects this signal to a 1K pullup resistor to generate a real logic level for this line. It's possible, though more complicated, to build a Zorro III PIC that can actually run in Zorro II mode when in a Zorro II backplane. It's hardly necessary or required to support this backward compatibility mechanism, and in many cases it'll be impractical. The Zorro III specification does require that this signal be used, at least, to shut the card down and pass /CFGIN to /CFGOUT when in a Zorro II backplane.

4.5 DMA Control Signals

There are various signals on the expansion bus that coordinate the arbitration of bus masters. Zorro II bus masters use some of the same logical signals, but their arbitration protocol is considerably different.

PIC is DMA Owner (/OWN)

This is asserted by the bus controller when a master is about to go on the bus and indicates that some master owns the bus. Zorro II bus masters drive this, and some Zorro III slaves may find a need to monitor it, or /BGACK, to determine who's the bus master. This is ordinarily not important to Zorro III PICs, and they may not drive this line.

Slot Specific Bus Arbitration (/BR_N, /BG_N)

These are the slot-specific /BR_N and /BG_N signals, where "N" refers to the expansion slot number. The bus request from each board is taken in by the bus controller and ultimately used to take over the system from the primary bus master, which is always the local master. Zorro III PICs toggle /BR_N to register or unregister as a master with the bus controller. /BG_N is asserted to one registered PIC at a time, on a cycle by cycle basis, to indicate to the PIC that it gets the bus for one full cycle.

Bus Grant Acknowledge (/BGACK)

Asserted by the bus controller when a master is about to go on the bus. As with /OWN, most Zorro III PICs ignore this signal, and none may drive it.

Bus Want/Clear (/BCLR)

This signal is asserted by the bus controller to indicate that a PIC wants to master the bus; Zorro III cards can use this to determine if any Zorro II bus requests are pending; Zorro III bus requests don't affect /BCLR.

4.6 Address and Related Control Signals

These signals are various items used for the addressing of devices in Zorro III mode by bus masters either on the bus or from the local bus. The bus controller translates local bus signals (68030 protocol on the A3000) into Zorro III signals; masters are responsible for creating the appropriate signals via their own bus control logic.

Read Enable (READ)

Read enable for the bus; READ asserted during a bus cycle indicates a read cycle, READ negated indicates a write cycle. READ is asserted at address time, prior to /FCS, for a full cycle, and prior to /MTCR for a short cycle. READ stays valid throughout the cycle; no latching required.

Multiplexed Address Bus (A₈-A₃₁)

These signals are driven by the bus master during address time, prior to the assertion of /FCS. Any responding slave must latch as many of these lines as it needs on the falling edge of /FCS, as they're tri-stated very shortly after /FCS goes low. These addresses always include all configuration address bits for normal cycles, and the cycle type information for Quick Interrupt cycles.

Short Address Bus (A₂-A₇)

These signals are driven by the bus master during address time, prior to the assertion of

/FCS, for full cycles, and prior to the assertion of /MTCR for short cycles. They stay valid for the entire full or short cycle, and as such do not need to be latched by responding slaves.

Memory Space (FC0-FC2)

The memory space bits are an extension to the bus address, indicating which type of access is taking place. Zorro III PICs must pay close attention to valid memory space types, as the space type can change the type of the cycle driven by the current bus master. The encoding is the same as the valid Motorola function codes for normal accesses. These are driven at address time, and like the low short address, are valid for an entire short or full cycle.

Table 4-1: Memory Space Type Codes

FC0	FC1	FC2	Address Space Type	Z3 Response
0	0	0	Reserved	None
0	0	1	User Data Space	Memory
0	1	0	User Program Space	Memory
0	1	1	Reserved	None
1	0	0	Reserved	None
1	0	1	Supervisor Data Space	Memory
1	1	0	Supervisor Program Space	Memory
1	1	1	CPU Space	Interrupts

Compatibility Cycle Strobe (/CCS)

This is equivalent to the Zorro II address strobe, /AS. A Zorro III PIC doesn't use this for normal operation, but may use it during the autoconfiguration process if configuring at the Zorro II address. AUTOCONFIGTM cycles at \$00E8xxxx always look like Zorro II cycles, though of course /FCS and the full Zorro III address is available, so a card can use either Zorro II or Zorro III addressing to start the cycle. However, using the /CCS strobe can save the designer the need to compare the upper 8 bits of address. Data must be driven Zorro II style, though if the /DSN lines are respected for reads, /CINH is asserted, and /MTACK is negated, the resulting Zorro III cycle will fit within the expected Zorro II cycle generated by the bus controller. Yes, that should sound weird; it's based on the mapping of Zorro II vs. Zorro III signals, and of course the fact that /FCS always starts any cycle. Also note that a bus cycle with /CCS asserted and /FCS negated is always a Zorro II PIC-as-master cycle. Many Zorro III cards will instead configure at the alternate \$FF00xxxx base address, fully in Zorro III mode, and thus completely ignore this signal.

Full Cycle Strobe (/FCS)

This is the standard Zorro III full cycle strobe. This is asserted by the bus master shortly after addresses are valid on the bus, and signals the start of any kind of Zorro III bus cycle. Shortly after this line is asserted, all the multiplexed addresses will go invalid, so in general, all slaves latch the bus address on the falling edge of /FCS. Also, /BGN line is negated for a Zorro III mastered cycle shortly after /FCS is asserted by the master.

4.7 Data and Related Control Signals

The data time signals here manage the actual transfer of data between master and slave for both full and short cycle types. The burst mode signals are here too, as they're basically data phase signals even though they don't only concern the transfer of data.

Data Output Enable (DOE)

This signal is used by an expansion card to enable the buffers on the data bus. The bus master drives this line to keep slave PICs from driving data on the bus until *data time*.

Data Bus (D0-D31)

This is the Zorro III data bus, which is driven by either the master or the slave when DOE is asserted by the master (based on READ). It's valid for reads when /DTACK is asserted by the slave; on writes when at least one of /DS_N is asserted by the master, for all cycle types.

Data Strobes (/DS_N)

These strobes fall during *data time*; /DS₃ strobes D₂₄-D₃₁, while /DS₀ strobes D₀-D₇. For write cycles, these lines signal data valid on the bus. At all times, they indicate which bytes in the 32 bit data word the bus master is actually interested in. For cachable reads, all four bytes must be returned, regardless of the value of the sizing strobes. For writes, only those bytes corresponding to asserted /DS_N are written. Only contiguous byte cycles are supported; e.g. /DS₃₋₀ = 2, 4, 5, 6, or 10 is invalid.

Data Transfer Acknowledge (/DTACK)

This signal is used to normally terminate a Zorro III cycle. The slave is always responsible for driving this signal. For a read cycle, it asserts /DTACK as soon as it has driven valid data onto the data bus. For a write cycle, it asserts /DTACK as soon as it's done with the data. Latching the data on writes may be a good idea; that can allow a slave to end the cycle before it has actually finished writing the data to its local memory.

Cache Inhibit (/CINH)

This line is asserted at the same time as /SLAVEN to indicate to the bus master that the cycle must not be cached. If a device doesn't support caching, it must assert /CINH and actually obey the /DS_N byte strobes for read cycles. Conversely, if the device supports caching, /CINH is negated and the device returns all four bytes valid on reads, regardless of the actual supplied /DS_N strobes.

Multiple Cycle Transfers (/MTCR,/MTACK)

These lines comprise the Multiple Transfer Cycle handshake signals. The bus master asserts /MTCR at the start of *data time* if it's capable of supporting Multiple Transfer Cycles, and the slave asserts /MTACK with /SLAVEN if it's capable of supporting Multiple Transfer Cycles. If the handshake goes through, /MTCR strobes in the short address and write data as long as the full cycle continues.

CHAPTER 5

TIMING

*"When dealing with the insane, the best method is
to pretend to be sane."*

-Hermann Hesse

ALL TIMING INFORMATION IS PRELIMINARY

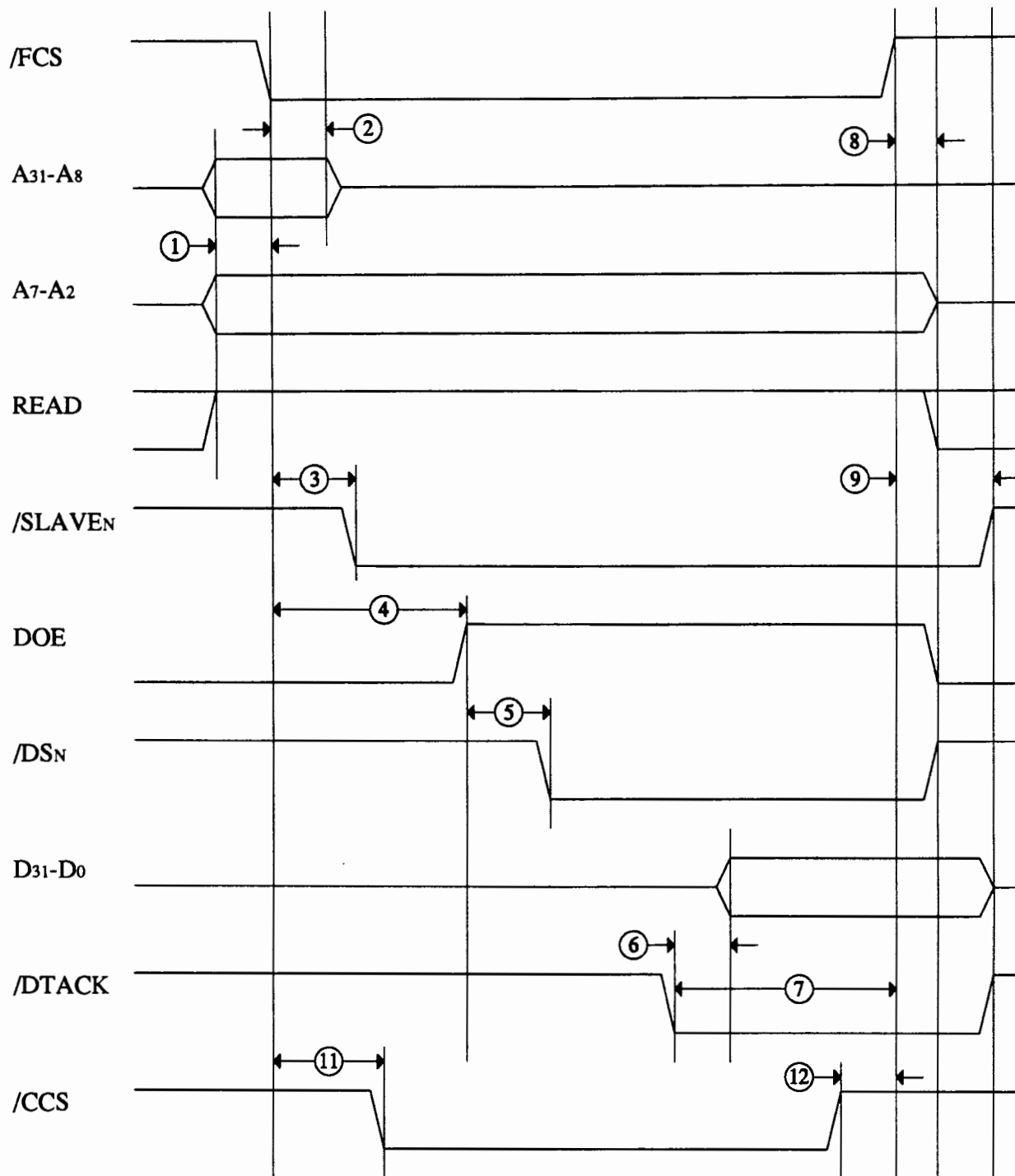
This information is all preliminary. Nothing is expected to get any more speed critical, but as mentioned previously, the testing of Zorro III designs has just started at the time of this writing, final bus controllers are not yet available, and only a few PIC designs have even been conceived.

This section covers the various timing specifications in detail for different Zorro III operations. It's important to realize that this timing information is a **specification**. Actual Zorro III systems may offer much more relaxed timings. Today. The whole point of the specification is that as long as all Zorro III PICs and all Zorro III backplanes base things on the timings given here, they'll always work together nicely. Any design based on the actual characteristics of any particular backplane will very likely wind up working only on that particular backplane.

The philosophy of timing on the Zorro III bus is to keep things as simple as possible without compromising the performance goals of the bus. Zorro III PICs are expected to be based on F-Series TTL logic, fast PALs, and possibly full custom chip designs. It's very unlikely the designer will meet any of these specifications with the LS parts left over from old Zorro II card designs.

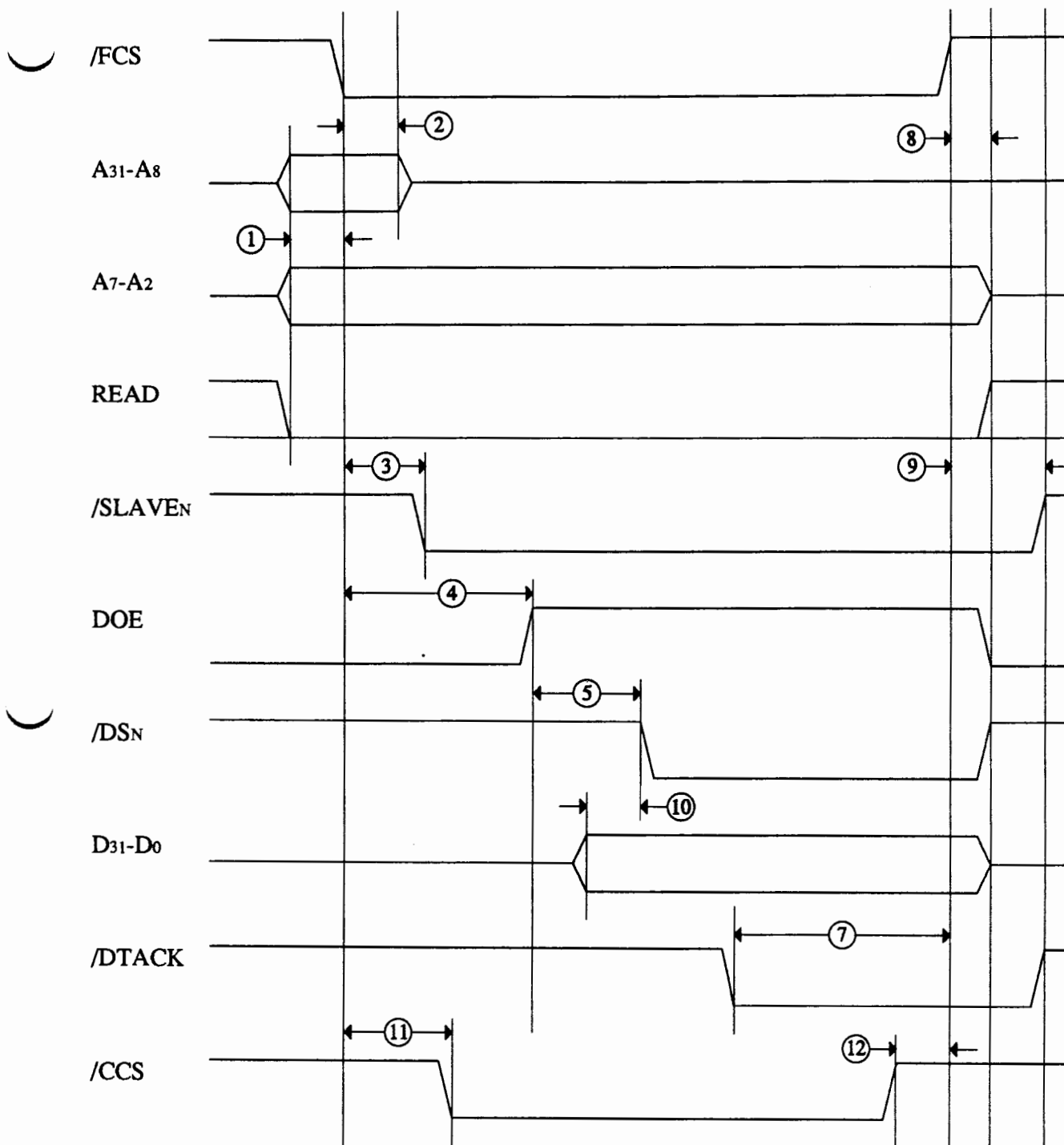
5.1 Standard Read Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N delay	T _{SLV}	-----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	-----
5	DOE to /DS _N delay	T _{Ds}	10ns	-----
6	Data setup to /DTACK	T _{RDS}	0ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
11	/FCS to /CCS delay	T _{CCS}	35ns	175ns
12	/CCS off to /FCS off	T _{ovL}	40ns	-----



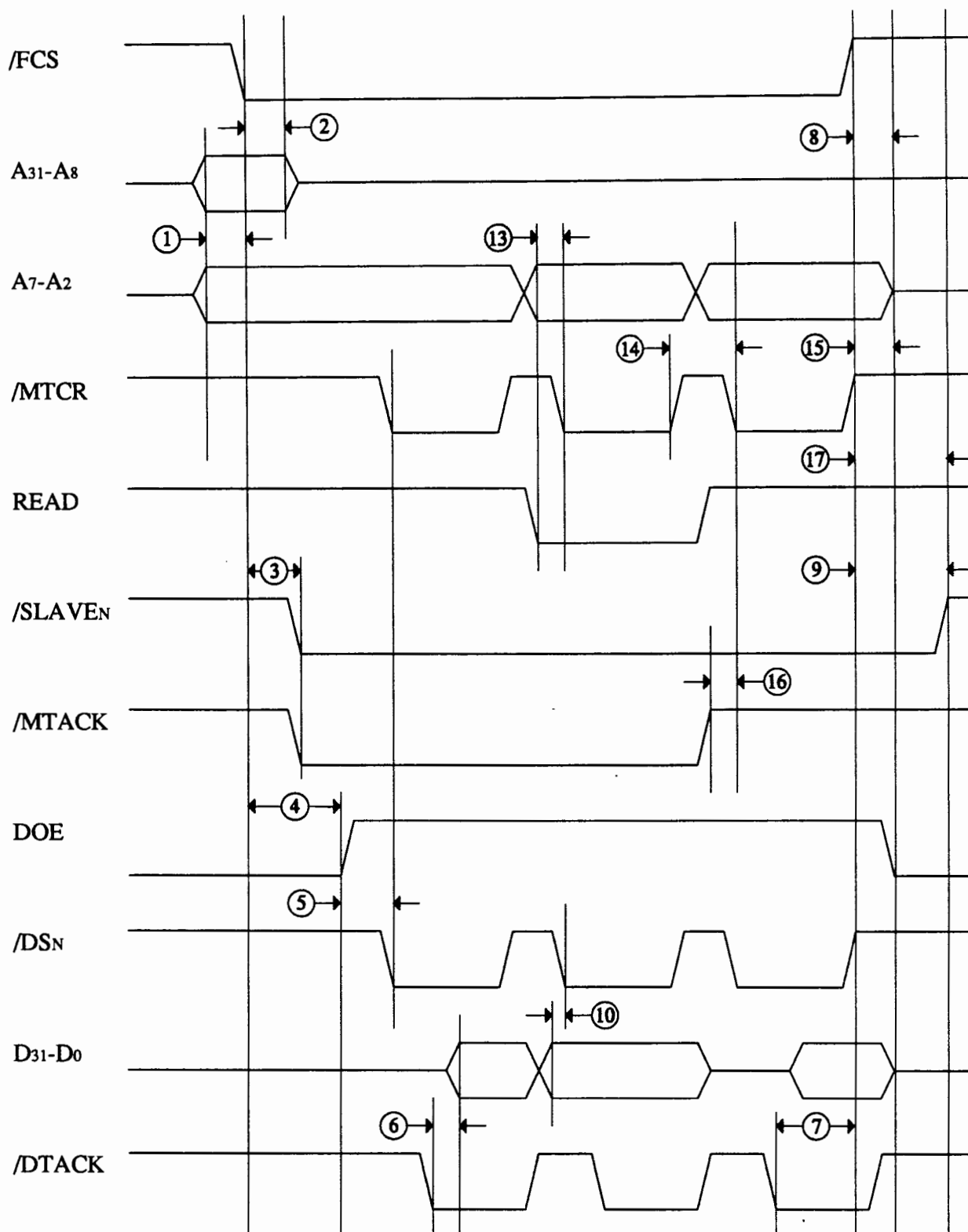
5.2 Standard Write Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N delay	T _{SLV}	-----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	-----
5	DOE to /DS _N delay	T _{DS}	10ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
10	Write data setup to /DS _N	T _{WDS}	5ns	-----
11	/FCS to /CCS delay	T _{CCS}	35ns	175ns
12	/CCS off to /FCS off	T _{ovL}	40ns	-----



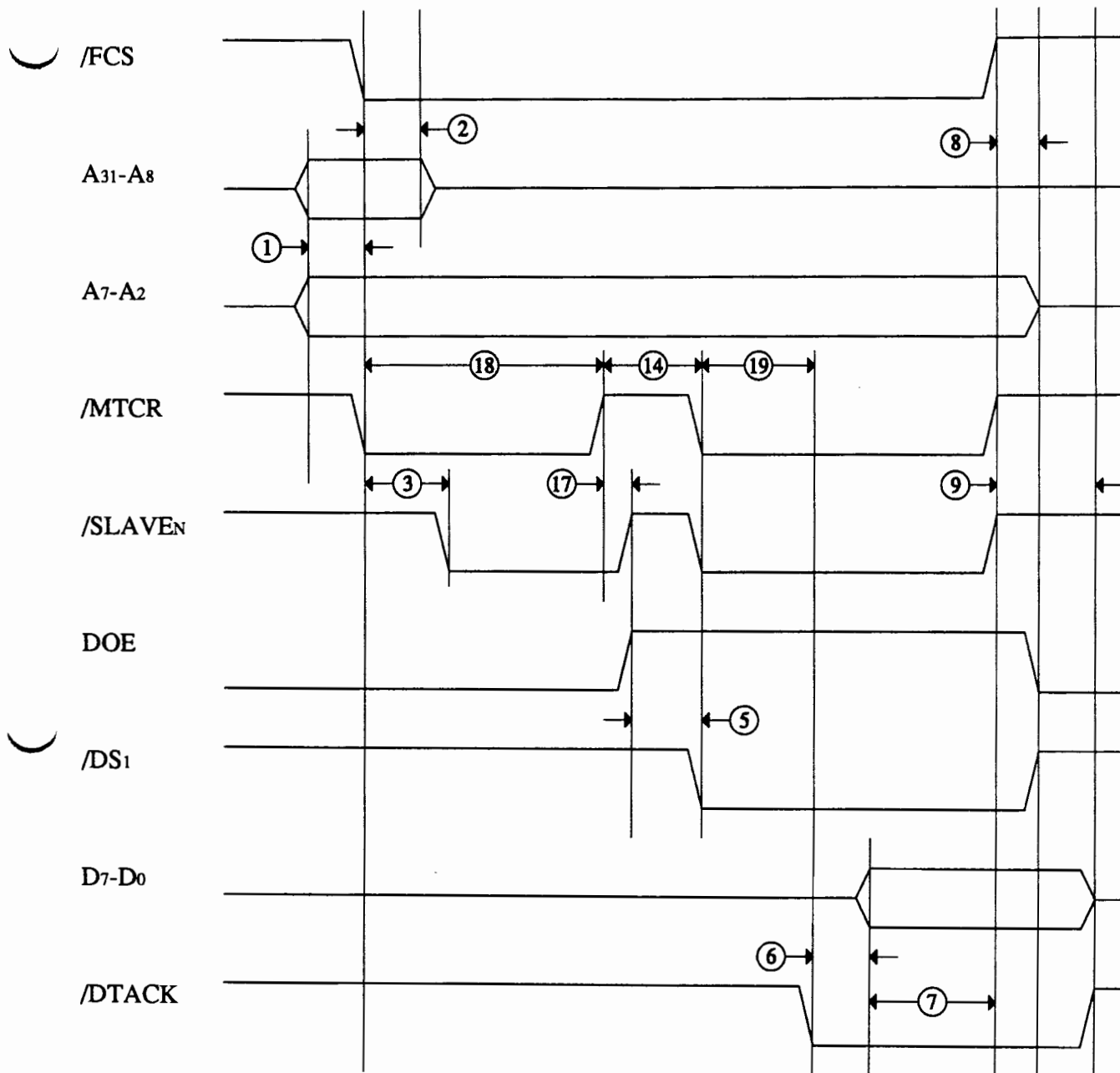
5.3 Multiple Transfer Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N , /MTACK delay	T _{SLV}	-----	25ns
4	/FCS to DOE delay	T _{DOE}	30ns	-----
5	DOE to /DS _N , /MTCR delay	T _{DS}	10ns	-----
6	Data setup to /DTACK	T _{RDS}	0ns	-----
7	/DTACK to /FCS, /MTCR off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
10	Write data setup to /DS _N	T _{WDS}	5ns	-----
13	Address, READ setup to /MTCR	T _{AMS}	5ns	-----
14	/MTCR off to /MTCR on	T _{REF}	10ns	-----
15	Address, READ hold from /MTCR	T _{HAM}	0ns	-----
16	/MTACK off to /MTCR	T _{BCD}	10ns	-----
17	Slave signal hold from /MTCR off	T _{HSM}	0ns	5ns



5.4 Quick Interrupt Cycle Timing

No.	Name	Symbol	Min	Max
1	Address setup to /FCS	T _{AFS}	15ns	-----
2	Address hold from /FCS	T _{HAF}	10ns	-----
3	/FCS to /SLAVE _N delay	T _{SLV}	-----	25ns
5	DOE to /DS _N delay	T _{DS}	10ns	-----
6	Data setup to /DTACK	T _{RDS}	0ns	-----
7	/DTACK to /FCS off	T _{OFF}	10ns	-----
8	Master signal hold from /FCS off	T _{HMC}	0ns	5ns
9	Slave signal hold from /FCS off	T _{HSC}	0ns	15ns
14	/MTCR off to /MTCR on	T _{REF}	10ns	-----
17	Slave signal hold from /MTCR off	T _{HSM}	0ns	5ns
18	Poll Phase time	T _{POL}	30ns	100ns
19	Vector Phase start to /DTACK time	T _{VEC}	-----	100ns



CHAPTER 6

Electrical Specifications

*"...I collected the instruments of life around me, that I might infuse a spark of
being into the lifeless thing that lay at my feet"*
-Victor Frankenstein

The Zorro III bus has a number of electrical specifications that are very important for PIC designers to consider, along with the timing parameters of course. It's extremely important to base designs on the specification of the backplane, rather than the actual behavior of the backplane. New backplanes for new machines are designed to conform to the specification, they are not necessarily based on previous designs. This is especially important with the Zorro III bus, since timing is far more critical than in the past, and the bus controller is designed from this specification, rather than the reverse, as in the Amiga 2000.

6.1 Expansion Bus Loading

The Zorro III bus loading is specified based on typical TTL family "F" series buffer devices, though in reality, compatible CMOS devices are likely to be used in some bus controllers or PICs. Thus, it's important to accept the TTL levels as a minimum voltage level, and make sure that all inputs are the appropriate TTL levels, while outputs can be at TTL or CMOS voltage levels as long as they provide the required source and sink.

While some A2000 designs used "LS" or "ALS" buffers instead of "F", the bus will generally work with these older cards, at least with current backplane designs such as the A3000 backplane. However, Zorro III designs must exactly obey these loading rules; it's very probable that some future Zorro III machines will have a large number of slots. In such machines, PICs built on the Zorro II specification will still work in a lightly loaded bus, but may not function in a fully loaded bus. All Zorro III PICs built to spec will work in any Zorro III backplane, without any loading problems, if all loading and timing rules are followed by the PIC designer. The bus

Table 6-1: Zorro III Drive Types

Signal	Direction	High Level	Low Level
Standard	Loading Driven	+140 μ A @ +2.7VDC +2.5VDC @ -3.0mA	-3.2mA @ +0.4VDC +0.4VDC @ +64mA
Clock	Loading	+20 μ A @ +2.7VDC	-1.6mA @ +0.4VDC
O.C.	Loading Driven	+80 μ A @ +2.7VDC Not Driven	-3.2mA @ +0.4VDC +0.4VDC @ +20mA
Non-bussed	Loading Driven	+80 μ A @ +2.7VDC +2.5VDC @ -0.4mA	-1.0mA @ +0.4VDC +0.4VDC @ +4.0mA

signals are divided up into the four groups shown in Table 6-1, based on the loading characteristics of the particular signal. The signals in each group are given here.

6.1.1 Standard Signals

The majority of signals on the bus are in this group. These are bussed signals, driven actively on the bus by F-series (or compatible) drivers such as 74F245, usually tri-stated when ownership of the signal changed for master and slave, and generally terminated with a 220 Ω /330 Ω thevenin terminator. PICs can apply two standard loads to each of these signals when necessary.

/FCS	/CCS	/DS ₀ -/DS ₃	/LOCK
A ₂ -A ₇	AD ₈ -AD ₃₁	SD ₀ -SD ₇	READ
FC ₀ -FC ₂	DOE	/IORST	/BCLR
/MTCR	/MTACK		

6.1.2 Clock Signals

All clock signals on the bus are in this group. Many designs are very sensitive to clock delay, skew, and rise/fall times, so loading on the clock lines must be kept to a minimum. These are bussed signals, actively driven by the backplane, and source terminated with a low value

series resistor. PICs can apply one standard load to each of these signals when necessary. Zorro II cards have the same clock rules, so there should never be clocking problems when using either card type in a backplane.

/C3	CDAC	/C1	7M
E Clock			

6.1.3 Open Collector Signals

Many of the bus signals are shared via open collector or open drain outputs rather than via tri-stated signals; this is of course required for some asynchronous things like the shared interrupt lines, and it works well for other types of signals as well. Of course, a backplane resistor pulls these lines high, PICs only drive the line low.

/OWN	/BGACK	/CINH	/BERR
/DTACK	/RESET	/INT ₁	/INT ₂
/INT ₄	/INT ₅	/INT ₆	/INT ₇
/HLT			

6.1.4 Non-bussed Signals

The non-bussed, or slot specific, signals are involved with only one slot on the bus (eg, each slot has its own copy). As a result, the drive requirements are much less for these signals. The backplane provides pullups or pulldowns, as required by the specific signal.

/CFGIN _N	/CFGOUT _N	/BR _N	/BG _N
SenseZ ₃	/SLAVEN		

6.2 Slot Power Availability

The system power for the Zorro III bus is totally based on the slot configurations. A backplane is always free to supply extra power, but it must meet the minimum requirements specified here. All PICs must be designed with the minimum specifications in mind, especially the tolerances.

Pin	Supply
5,6	+5 VDC \pm 5% @ 2 Amps
8	-5 VDC \pm 5% @ 60 mA
10	+12 VDC \pm 5% @ 500mA
20	-12 VDC \pm 5% @ 60mA

6.3 Temperature Range

The Zorro III bus is specified for operation over a temperature range of 0° C to 70° C.

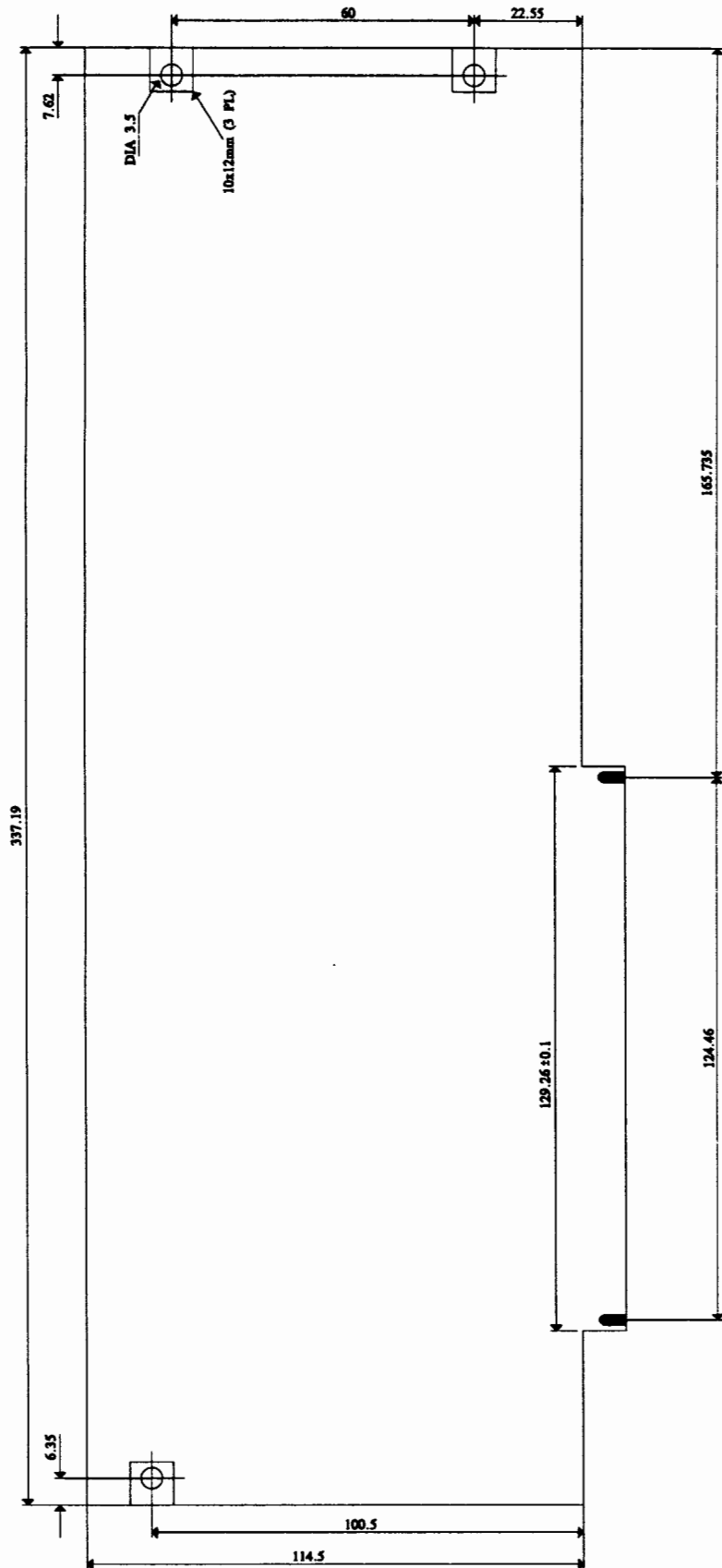
CHAPTER 7

MECHANICAL SPECIFICATIONS

"Never speak more clearly than you think."

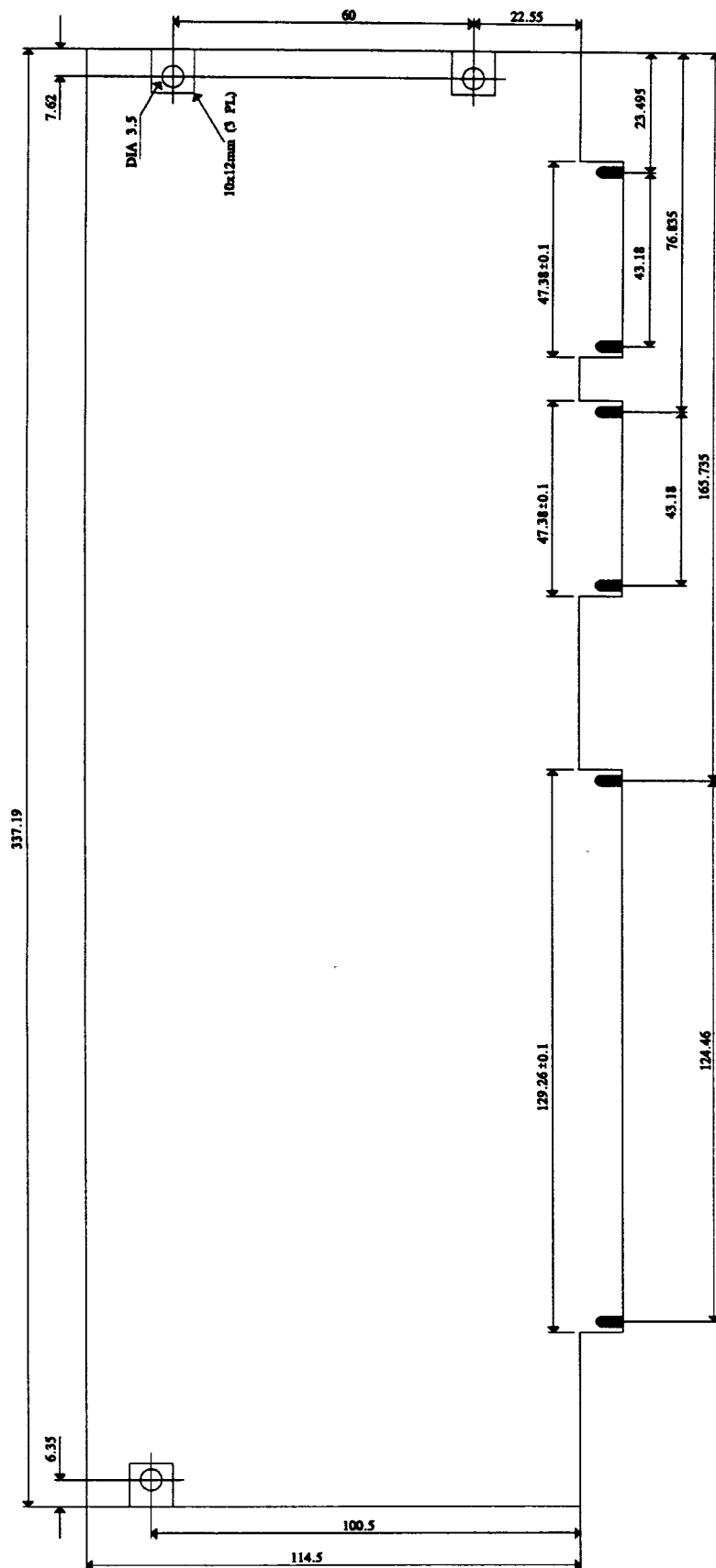
-Jeremy Bernstein

This section covers the various mechanical details of Zorro III cards. Note that these specifications are considered preliminary.



7.1 Basic Zorro III PIC

This drawing shows the basic Zorro III PIC. All of the dimensions are in millimeters.



7.3 PIC with Video Option

This drawing shows the basic Zorro III PIC, with both Zorro III and the Amiga Video Slot fingers specified. All of the dimensions are in millimeters. Please consult the *A500/A2000 Technical Reference Manual* for the form factor specification of a video-only card that will fit both Amiga 2000 and Amiga 3000 computers.

CHAPTER 8

AUTOCONFIG™

"The goal of all inanimate objects is to resist man and ultimately defeat him."

-Russell Baker

8.1 The AUTOCONFIG™ Mechanism

The AUTOCONFIG™ mechanism used for the Zorro III bus is an extension of the original Zorro II configuration mechanism. The main reason for this is that the Zorro II mechanism works so well, there was little need to change anything. The changes are simply support for new hardware features on the Zorro III bus.

Amiga autoconfiguration is surprisingly simple. When an Amiga powers up or resets, every card in the system goes to its unconfigured state. At this point, the most important signals in the system are /CFGIN_N and /CFGOUT_N. As long as a card's /CFGIN_N line is negated, that card sits quietly and does nothing on the bus (though memory cards should continue to refresh even through reset, and any local board activities that don't concern the bus may take place after /RESET is negated). As part of the unconfigured state, /CFGOUT_N is negated by the PIC immediately on reset.

The configuration process begins when a card's /CFGIN_N line is asserted, either by the backplane, if it's the first slot, or via the configuration chain, if it's a later card. The configuration chain simply ensures that only one unconfigured card will see an asserted /CFGIN_N at one time. An unconfigured card that sees its /CFGIN_N line asserted will respond to a block of memory called *configuration space*. In this block, the PIC will assert a set of read-only

registers, followed by a set of write-only registers (the read-only registers are also known as AUTOCONFIG™ ROM). Starting at the base of this block, the read registers describe the device's size, type, and other requirements. The operating system reads these, and based on them, decides what should be written to the board. Some write information is optional, but a board will always be assigned a base address or be told to shut up. The act of writing the final bit of base address, or writing anything to a shutup address, will cause the PIC to assert its /CFGOUTN, enabling the next board in the configuration chain.

The Zorro II configuration space is the 64K memory block \$00E8xxxx, which of course is driven with 16 bit Zorro II cycles; all Zorro II cards configure there. The Zorro III configuration space is the 64K memory block beginning at \$FF00xxxx, which is always driven with 32 bit Zorro III cycles (PICs need only decode A31-A24 during configuration). A Zorro III PIC can configure in Zorro II or Zorro III configuration space, at the designer's discretion, but not both at once. All read registers physically return only the top 4 bits of data, on D31-D28 for either bus mode. Write registers are written to support nybble, byte, and word registers for the same register, again based on what works best in hardware. This design attempts to map into real hardware as simply as possible. Every AUTOCONFIG™ register is logically considered to be 8 bits wide; the 8 bits actually being nybbles from two paired addresses.

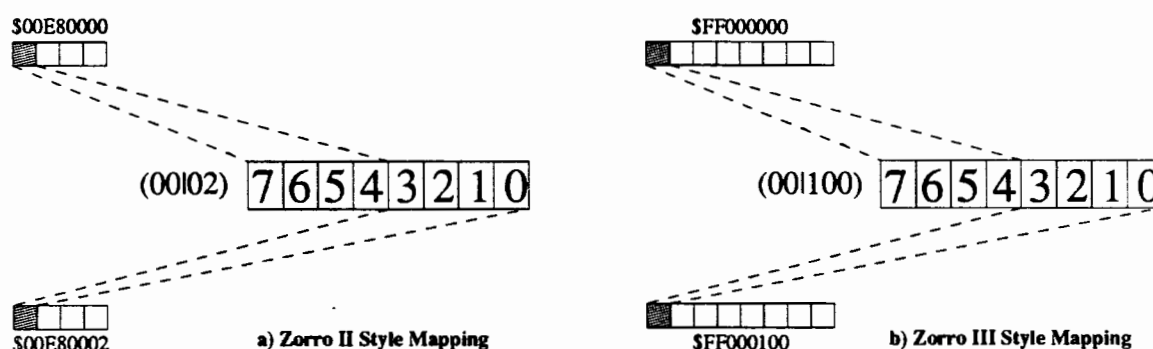


Figure 8-1: Configuration Register Mapping

The register mappings for the two different blocks are shown in Figure 8-1. All the bit patterns mentioned in the following sections are logical values. To avoid ambiguity, all registers are referred to by the number of the first register in the pair, since the first pair member is the same for both mapping schemes. In the actual implementation of these registers, all read registers except for the 00 register are physically complemented; eg, the logical value of register 3C is always 0, which means in hardware, the upper nybbles of locations \$00E8003C and \$00E8003E, or \$FF00003C and \$FF00003E, both return all 1's.

8.2 Register Bit Assignments

The actual register assignments are below. Most of the registers are the same as for the Zorro II bus, but are included here anyway for completeness. The Amiga OS software names for these registers in the ExpansionRom or ExpansionControl structures are included.

Reg Z2 Z3 Bit

00	02 100 (er_Type)	7,6	These bits encode the PIC type:	
			00	Reserved
			01	Reserved
			10	Zorro III
			11	Zorro II
		5	If this bit is set, the PIC's memory will be linked into the system free pool. The Zorro III register 08 may modify the size of the linked memory.	
		4	Setting this bit tells the OS to read an autoboot ROM.	
		3	This bit is set to indicate that the next board is related to this one; often logically separate PICs are physically located on the same card.	
		2-0	These bits indicate the configuration size of the PIC. This size can be modified for the Zorro III cards by the size extension bit, which is the new meaning of bit 5 in register 08.	
			Bits	Unextended Extended
			000	8 megabytes 16 megabytes
			001	64 kilobytes 32 megabytes
			010	128 kilobytes 64 megabytes
			011	256 kilobytes 128 megabytes
			100	512 kilobytes 256 megabytes
			101	1 megabyte 512 megabytes
			110	2 megabytes 1 gigabyte
			111	4 megabytes RESERVED
04	06 104 (er_Product)	7-0	The device's product number, which is completely up to the manufacturer. This is generally unique between different products, to help in identification of system cards, and it must be unique between devices using the automatic driver binding features.	
08	0A 108 (er_Flags)	7	This was originally an indicator to place the card in the 8 megabyte Zorro II space, when set, or anywhere it'll fit, if cleared. Under the Zorro III spec, this is set to indicate that the board is basically a memory device, cleared to indicate that the board is basically an I/O device.	
		6	This bit is set to indicate that the board can't be shut up by software, cleared to indicate that the board can be shut up.	
		5	This is the size extension bit. If cleared, the size bits in register 00 mean the same as under Zorro II, if set, the size bits indicate a new size. The	

most common new Zorro III sizes are the smaller ones; all new sized cards get aligned on their natural boundaries.

- 4 For OS 1.3 compatibility, must be 1.
- 3-0 These bits indicate a board's sub-size; the amount of memory actually used/required by a PIC. For memory boards that auto-link, this is the actual amount of memory that will be linked into the system free memory pool. A memory card, with memory starting at the base address, can be automatically sized by the Operating System. This sub-size option is intended to support cards with variable setups without requiring variable physical configuration capability on such cards. It also may greatly simplify a Zorro III design, since 16 megabyte cards and up can be designed with a single latch and comparator for base address matching, while 8 megabyte and smaller PICs require large latch/comparator circuits not available in standard TTL packages.

Bits	Encoding
0000	Logical size matches physical size
0001	Automatically sized by the Operating System
0010	64 kilobytes
0011	128 kilobytes
0100	256 kilobytes
0101	512 kilobytes
0110	1 megabyte
0111	2 megabytes
1000	4 megabytes
1001	6 megabytes
1010	8 megabytes
1011	10 megabytes
1100	12 megabytes
1101	14 megabytes
1110	Reserved
1111	Reserved

For boards that wish to be automatically sized by the operating system, a few rules apply. The memory is sized in 512K increments, and grows from the base address upward. Memory wraps are detected, but the design must ensure that its data bus doesn't float when the sizing routine addresses memory locations that aren't physically present on the board; data bus pullups or pulldowns are recommended. This feature is designed to allow boards to be easily upgraded with additional or increased density memories without the need for memory configuration jumpers.

Reg Z2 Z3 Bit

0C	0E 10C 7-0 (er_Reserved03)	Reserved, must be 0.
10	12 110 7-0	Manufacturer's number, high byte.
14	16 114 7-0 (er_Manufacturer)	Manufacturer's number, low bytes. These are unique, and can only be assigned by Commodore.
18	1A 118 7-0	Optional serial number, byte 0 (msb)
1C	1E 11C 7-0	Optional serial number, byte 1
20	22 120 7-0	Optional serial number, byte 2
24	26 124 7-0 (er_SerialNumber)	Optional serial number, byte 3 (lsb) This is for the manufacturer's use and can contain anything at all. The main intent is to allow a manufacturer to uniquely identify individual cards, but it can certainly be used for revision information or other data.
28	2A 128 7-0	Optional ROM vector, high byte.
2C	2E 12C 7-0 (er_InitDiagVec)	Optional ROM vector, low byte. If the ROM address valid bit (bit 4 of register (00102)) is set, these two registers provide the sixteen bit offset from the board's base at which the start of the ROM code is located. If the ROM address valid bit is cleared, these registers are ignored.
30	32 130 7-0 (er_Reserved0c)	Reserved, must be 0. Unsupported base register reset register under Zorro II*.
34	36 134 7-0 (er_Reserved0d)	Reserved, must be 0.
38	3A 138 7-0 (er_Reserved0e)	Reserved, must be 0.
3C	3E 13C 7-0 (er_Reserved0f)	Reserved, must be 0.
40	42 140 7-0 (ec_Interrupt)	Reserved, must be 0. Unsupported control state register under Zorro II*.
44	46 144 7-0	High order base address register, write only.
48	4A 148 7-0 (ec_Z3_HighByte) (ec_BaseAddress)	Low order base address register, write only. The high order register takes bits 31-24 of the board's configured address, the low ordered resgister takes bits 23-16. For Zorro III boards configured in the Zorro II space, the configuration address is written both nybble and byte wide, with the ordering:

*The original Zorro specifications called for a few registers, like these, that remained active after configuration. Support for this is impossible, since the configuration registers generally disappear when a board is configured, and absolutely must move out of the \$00E8xxx space. So since these couldn't really be implemented in hardware, system software has never supported them. They're included here for historical purposes.

Reg Z2 Z3 Bit

Reg	Nybble	Byte
46	A27-A24	N/A
44	A31-A28	A31-A24
4A	A19-A16	N/A
48	A23-A20	A23-A16

Note that writing to register 48 actually configures the board for both Zorro II and Zorro III boards in the Zorro II configuration block. For Zorro III PICs in the Zorro III configuration block, the action is slightly different. The software will actually write the configuration as byte and word wide accesses:

Reg	Byte	Word
48	A23-A16	N/A
44	A31-A24	A31-A16

The actual configuration takes place when register 44 is written, thus supporting any physical size of configuration register.

4C	4E 14C 7-0 (ec_Shutup)	Shut up register, write only. Anything written to 4C will cause a board that supports shut-up to completely disappear until the next reset.
50	52 150 7-0	Reserved, must be 0.
54	56 154 7-0	Reserved, must be 0.
58	5A 158 7-0	Reserved, must be 0.
5C	5E 15C 7-0	Reserved, must be 0.
60	62 160 7-0	Reserved, must be 0.
64	66 164 7-0	Reserved, must be 0.
68	6A 168 7-0	Reserved, must be 0.
6C	6E 16C 7-0	Reserved, must be 0.
70	72 170 7-0	Reserved, must be 0.
74	76 174 7-0	Reserved, must be 0.
78	7A 178 7-0	Reserved, must be 0.
7C	7E 17C 7-0	Reserved, must be 0.

APPENDICES

"I have been given the freedom to do as I see fit."

-REM

A.1 Physical and Logical Signal Names

The Amiga 3000 Bus signals vary based on the particular bus mode in effect. This table lists each physical pin by physical name, and then by the logical names for Zorro II mode, Zorro III mode, address phase, and Zorro III data mode, data phase.

PIN NO.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
1	Ground	Ground	Ground	Ground
2	Ground	Ground	Ground	Ground
3	Ground	Ground	Ground	Ground
4	Ground	Ground	Ground	Ground
5	+5VDC	+5VDC	+5VDC	+5VDC
6	+5VDC	+5VDC	+5VDC	+5VDC
7	/OWN	/OWN	/OWN	/OWN
8	-5VDC	-5VDC	-5VDC	-5VDC
9	/SLAVEN	/SLAVEN	/SLAVEN	/SLAVEN
10	+12VDC	+12VDC	+12VDC	+12VDC
11	/CFGOUTN	/CFGOUTN	/CFGOUTN	/CFGOUTN
12	/CFGINN	/CFGINN	/CFGINN	/CFGINN
13	Ground	Ground	Ground	Ground
14	/C3	/C3 Clock	/C3 Clock	/C3 Clock
15	CDAC	CDAC Clock	CDAC Clock	CDAC Clock
16	/C1	/C1 Clock	/C1 Clock	/C1 Clock
17	/CINH	/OVR	/CINH	/CINH
18	/MTCR	XRDY	/MTCR	/MTCR
19	/INT2	/INT2	/INT2	/INT2
20	-12VDC	-12VDC	-12VDC	-12VDC
21	A5	A5	A5	A5
22	/INT6	/INT6	/INT6	/INT6
23	A6	A6	A6	A6
24	A4	A4	A4	A4
25	Ground	Ground	Ground	Ground
26	A3	A3	A3	A3
27	A2	A2	A2	A2
28	A7	A7	A7	A7
29	/LOCK	A1	/LOCK	/LOCK
30	AD8	A8	A8	D0
31	FC0	FC0	FC0	FC0
32	AD9	A9	A9	D1
33	FC1	FC1	FC1	FC1
34	AD10	A10	A10	D2
35	FC2	FC2	FC2	FC2
36	AD11	A11	A11	D3
37	Ground	Ground	Ground	Ground
38	AD12	A12	A12	D4
39	AD13	A13	A13	D5
40	/INT7	(/EINT7)	/INT7	/INT7
41	AD14	A14	A14	D6
42	/INT5	(/EINT5)	/INT5	/INT5

PIN NO.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
43	AD15	A15	A15	D7
44	/INT4	(/EINT4)	/INT4	/INT4
45	AD16	A16	A16	D8
46	/BERR	/BERR	/BERR	/BERR
47	AD17	A17	A17	D9
48	/MTACK	(/VPA)	/MTACK	/MTACK
49	Ground	Ground	Ground	Ground
50	E Clock	E Clock	E Clock	E Clock
51	/DS0	(/VMA)	/DS0	/DS0
52	AD18	A18	A18	D10
53	/RESET	/RST	/RESET	/RESET
54	AD19	A19	A19	D11
55	/HLT	/HLT	/HLT	/HLT
56	AD20	A20	A20	D12
57	AD22	A22	A22	D14
58	AD21	A21	A21	D13
59	AD23	A23	A23	D15
60	/BR _N	/BR _N	/BR _N	/BR _N
61	Ground	Ground	Ground	Ground
62	/BGACK	/BGACK	/BGACK	/BGACK
63	AD31	D15	A31	D31
64	/BG _N	/BG _N	/BG _N	/BG _N
65	AD30	D14	A30	D30
66	/DTACK	/DTACK	/DTACK	/DTACK
67	AD29	D13	A29	D29
68	READ	READ	READ	READ
69	AD28	D12	A28	D28
70	/DS2	/LDS	/DS2	/DS2
71	AD27	D11	A27	D27
72	/DS3	/UDS	/DS3	/DS3
73	Ground	Ground	Ground	Ground
74	/CCS	/AS	/CCS	/CCS
75	SD0	D0	N/A	D16
76	AD26	D10	A26	D26
77	SD1	D1	N/A	D17
78	AD25	D9	A25	D25
79	SD2	D2	N/A	D18
80	AD24	D8	A24	D24
81	SD3	D3	N/A	D19
82	SD7	D7	N/A	D23
83	SD4	D4	N/A	D20
84	SD6	D6	N/A	D22

PIN NO.	Physical Name	Zorro II Name	Zorro III Address Phase	Zorro III Data Phase
85	Ground	Ground	Ground	Ground
86	SD ₅	D ₅	N/A	D ₂₁
87	Ground	Ground	Ground	Ground
88	Ground	Ground	Ground	Ground
89	Ground	Ground	Ground	Ground
90	Ground	Ground	Ground	Ground
91	SenseZ ₃	Ground	SenseZ ₃	SenseZ ₃
92	7M	E7M	7M	7M
93	DOE	DOE	DOE	DOE
94	/IORST	/BUSRST	/IORST	/IORST
95	/BCLR	/GBG	/BCLR	/BCLR
96	/INT ₁	(/EINT ₁)	/INT ₁	/INT ₁
97	/FCS	No Connect	/FCS	/FCS
98	/DS ₁	No Connect	/DS ₁	/DS ₁
99	Ground	Ground	Ground	Ground
100	Ground	Ground	Ground	Ground

A.2 A Glossary of Terms

The reader may be unfamiliar with a number of terms used in this document. Every effort has been made to include all such terms here.

address	A byte-numbered memory location. The Zorro II bus is based on a 24 bit address, the Zorro III bus on a 32 bit address.
arbitration	The unambiguous selection of one request out of a number of possible simultaneous requests for a resource. There are two kinds of arbitration in a Zorro III system; bus arbitration and quick interrupt arbitration.
asserted	The active state of a state, regardless of its logic sense.
atomic cycle	A cycle or set of cycles that are uninterruptable, and thus treated as a unit; both Multiple Transfer and LOCKed cycles are considered atomic under the Zorro III bus.
AUTOCONFIG™	From "automatic configuration", the Zorro bus specification for how software and hardware cooperate to permit PIC addresses to be set by software and PIC type information to be determined by software. This is explained in Chapter 8, and in the <i>A500/A2000 Technical Reference Manual</i> , available from Commodore-Amiga.
backplane	The cage or motherboard subsection into which PICs are inserted. The Amiga 2000 and Amiga 3000 computers have integral backplanes, the Amiga 500 and Amiga 1000 computers require add-on backplane cages for Zorro II compatibility.
burst	A short name for Multiple Transfer Cycle mode. Essentially, within one full Zorro III cycle there can be any number of Multiple Transfer Cycles. Each full cycle has a complete 32 bit address supplied and a complete 32 bit datum transferred. Each burst cycle supplies only the 8 bit page address, but transfers a complete 32 bit datum faster than the standard full cycle would allow.
bus cycle	One complete bus transaction, indicated by the assertion of least one cycle strobe. For any single bus cycle, there is one address, one data value, one data direction, and one cycle type in effect.
bus hogging	When a bus master takes over the bus for an undue amount of time. The Zorro II bus leaves it completely up to the individual PIC to avoid bus hogging; the Zorro III bus schedules PICs with the bus controller to evenly distribute the bus load.

bus starvation	When a master can't get access to the bus, it is said to be starved. On the Zorro II bus, two busy masters can completely starve a third master. Complete starvation is impossible on the Zorro III bus, though a bus hogging Zorro II card can cause similar symptoms.
byte	A collection of eight signals into a logical group, and the smallest independently addressable quantity on the Zorro bus.
clock	A free running signal driven at a fixed frequency to the bus, used mainly for clocking state machines on Zorro II cards.
cool	An unreachable goal for some, a way of life for others. The obvious example of this latter category being Dave Haynie, pictured at right.
cycle strobe	A bus signal that defines the boundary of a bus cycle; the Zorro II and Zorro II modes on a Zorro III bus each have their own cycle strobes. The current bus master always asserts the cycle strobes.
data	The contents of a memory location. The main purpose of a bus cycle is to transfer data between two locations. The Zorro II bus is based on a 16 bit data path, the Zorro III bus is based on a 32 bit data path.
DMA	Direct Memory Access; devices that have direct access to Zorro III slaves are said to have DMA capability. These devices are also called masters.
DMA latency	This is the time between a bus request and a bus grant as seen by a PIC wishing to become bus master.
device	A PIC, eg, a Zorro bus master or bus slave.
grant	The result of an arbitrated set of requests is a single grant; there are grants given for both the bus and quick interrupts.
Guinness	Attitude adjustment tonic, from Ireland. Said by some to be vital for sanity, if not normal human life.
hidden cycles	Cycles that occur on the local bus of a system, but can't be seen by devices on the expansion bus.
high	A signal driven to a logical +5V state is said to be high.
interrupt	An asynchronous line driven by a PIC to notify the CPU of some event, usually some hardware event governed by that PIC.



local bus	The main system bus of an Amiga computer is called the local bus. In general, the main CPU, video chips, chip memory, and any other built-in resources are on the local bus. The bus controller sits on both the local and expansion buses and manages the communications between them.
longword	Based on the Motorola conventions, a longword is equal to 4 bytes.
low	A signal driven to a logical +0V state is said to be low.
master	The device currently generating addresses for the expansion bus. There is only one master on the bus at a time, this being insured by the bus arbitration logic. The master also drives data on writes, the read, cycle, and data strobes, and several other signals.
motherboard	The main system circuit board for any Amiga computer. Resources on the local bus of a machine are often called motherboard resources.
negated	The inactive state of a signal, regardless of its logic sense.
nybble	A collection of four bits; one half of a byte. AUTOCONFIG™ ROMs are physically nybble-wide.
paragraph	A sequence of closely related sentences, generally expressing and supporting one succinct idea. This term has no special computerese meaning in any rational, modern system.
PIC	Plug In Card. Any Amiga expansion card is called a PIC for short.
request	Asking for the use of some resource; the Zorro III bus has two kinds of requests, bus requests and quick interrupt requests.
slave	The device currently responding to the address on the expansion bus. There is only one slave on the bus at a time; an error is signalled by the bus collision detect logic if multiple slaves respond to the same address. The slave also drives data on reads, the transfer acknowledge strobe, and several other signals.
slot	A physical port on a Zorro backplane, which supplies independent /SLAVEN, /BRN, and /BGN lines, chained /CFGIN _N and /CFGOUT _N lines, and is mechanically manifested as a 100 pin single-piece connector.
termination	Circuitry attached to a bus signal in order to minimize annoying analog things like ringing, reflections, crosstalk, and possibly random logic conditions which can arise when a bus is undriven.

timeout	A bus cycle terminated by the bus controller instead of by a responding slave device. If no slave responds to a bus cycle within a reasonable time period, the bus controller will terminate the cycle to prevent lockup of the system.
tri-state	A signal driven to a high impedance condition is said to be tri-stated.
word	Based on the Motorola conventions, a word is equal to 2 bytes.
Zorro	The name given to the Amiga bus specification. "Zorro I" refers to the original design for A1000 backplane boxes, "Zorro II" refers to the modification to this specification used for the A2000 and compatible backplanes, and "Zorro III" refers to the Zorro II compatible bus specification first used in the Amiga 3000 computer.

A.3 Zorro III Implementations

There aren't actually variable implementation levels supported for the Zorro III bus; all of the features are required for compliance with this specification. However, current prototype Amiga 3000 computers support only a subset of the Zorro III specification published here. This is, however, upgradable.

The A3000 implementation of the Zorro III bus is driven by a custom controller chip called **Fat Buster**. The specification of this chip and the A3000 hardware are fully capable of supporting the complete Zorro III bus, but the initial silicon on Fat Buster, called the Level 1 Fat Buster, omits some features. Missing are:

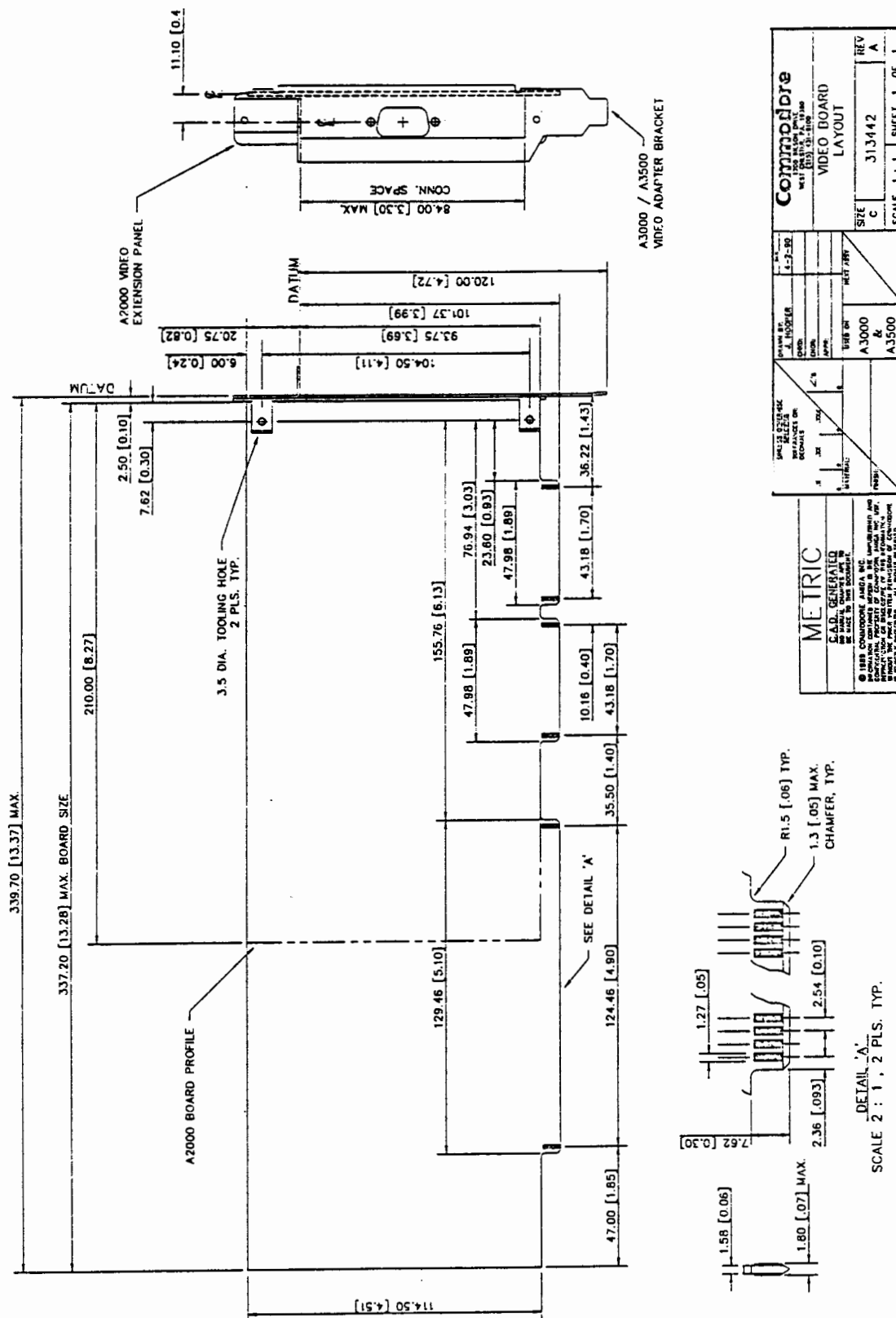
- Support of Multiple Transfer Cycles.
- Support for Zorro III style bus arbitration.
- Support for Quick Interrupts.

The Level 2 version of Fat Buster is currently under development at Commodore in West Chester, PA. and should be available very soon. Any developers who immediately intend to design PICs supporting these features are urged to contact Commodore Amiga Technical Support/Amiga Developer Support Europe for more information on obtaining samples of this part for use in A3000 Beta Test systems.

A3000 Video Board Form Factor

The diagram on the next page gives the form factor for video boards for the Amiga 3000. The A3000 video slot is the 72-pin connector closest to the rear of the motherboard. It is in-line with the 100-pin Zorro III Expansion Bus connector.

The schematic also shows the A2000 video board profile for those designing video cards that are plug-compatible with both the Amiga 3000 and 2000 models.





BIGRAM 8/32

A Complete Zorro III PIC
Design Example

Document Revision 1.00

Atlanta DevCon Release

by Dave Haynie

June 10, 1990

Copyright © 1990 Commodore-Amiga, Inc.

(

(

(

IMPORTANT INFORMATION

"We don't know a millionth of one percent about anything."

-Thomas Alva Edison

This Document Contains Preliminary Information

The information contained here, while a honest attempt to illustrate a good Zorro III card design, is still preliminary in nature and subject to possible errors and omissions. At the time of this writing, some of the features of this design were not yet testable in an Amiga 3000, as the enhanced bus controller chip was not yet available. We don't expect any problems with this design, but it's only responsible to supply you with this caveat.

Commodore Technology reserves the right to correct any mistake, error, omission, or viscious lie. Corrections will be published as updates to this document, which will be released as necessary in as developer-friendly a manner as possible. Revisions will be tracked via the revision number that appears on the front cover.

All information herein is Copyright © 1990 by Commodore-Amiga, Inc., and may not be reproduced in any form without permission.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	Intended Audience.....	1-1
1.2	A Few Words About AUTOCONFIG.....	1-2
1.3	Design Example Goals.....	1-2
CHAPTER 2	AUTOCONFIG™ LOGIC DESIGN	
2.1	Bus Buffers.....	2-1
2.2	The AUTOCONFIG ROM.....	2-2
2.3	The AUTOCONFIG Registers.....	2-4
2.4	The SLAVE Logic.....	2-5
CHAPTER 3	MEMORY SYSTEM DESIGN	
3.1	DRAM Refresh.....	3-1
3.1.1	Refresh Arbitration.....	3-2
3.1.2	Refresh Counter.....	3-2
3.1.3	Refresh Cycle.....	3-3
3.2	DRAM Access.....	3-4
3.2.1	Memory Cycle.....	3-5
3.2.2	Bank Selection.....	3-5
3.2.3	Address Multiplexing.....	3-6
CHAPTER 4	GOING FURTHER	
4.1	Designed-In Enhancements.....	4-1
4.1.1	The Experimenter's Board?.....	4-2

4.1.2	Multiple Cycle Transfer Support.....	4-2
4.2	Modification Ideas.....	4-2
4.2.1	Tighter RAM Cycles.....	4-3
4.2.2	Read/Write Optimizations.....	4-3
4.2.3	Standard DRAM Tricks.....	4-4

CHAPTER 5 ADDITIONAL ZORRO III ADVICE

5.1	Watch Those Synchronizations.....	5-1
5.2	Design for Speed.....	5-2
5.3	Follow the Specifications.....	5-3

APPENDICES

A.1	PAL Equations.....	A-1
A.1.1	Autoconfiguration Control PAL.....	A-2
A.1.2	Board Control PAL.....	A-4
A.1.3	Memory Timing PAL.....	A-6
A.1.4	CAS Control PAL.....	A-8
A.1.5	Refresh Counter PAL.....	A-10
A.2	Schematics.....	A-12
A.3	Zorro III Configuration.....	A-18

TABLES AND FIGURES

Table	2-1	Logical AUTOCONFIG Registers.....	2-2
Table	2-2	Physical ROM Registers.....	2-3
Figure	3-1	Refresh Arbitration.....	3-2
Figure	3-2	Refresh Cycle.....	3-3
Figure	3-3	Memory Access.....	3-4

—

—

—

CHAPTER 1

INTRODUCTION

*"The curtain rises on a vast primitive wasteland,
not unlike certain parts of New Jersey."*

-Woody Allen

This document fully describes an example Zorro III Plug-In-Card (PIC) design for a simple asynchronous dynamic RAM memory card. Its intent is to describe the procedures and underlying theories behind a basic Zorro III design. However, it is not a Zorro III designer's bible or any such rulebook. It should provide the designer with a better understanding of Zorro III PIC design, and perhaps provide a starting point for the beginning Amiga peripheral designer.

1.1 Intended Audience

This document was written primarily for hardware engineers interested in designing Plug-In-Cards for the Zorro III expansion bus. A reasonable level of microcomputer knowledge is a prerequisite to get much meaning out of these pages. A good understanding of the Zorro III bus theory, as outlined in *The Zorro III Bus Specification* (available from Commodore), is essential. Knowledge of basic TTL digital design with standard MSI and PAL devices is required, as is an understanding of dynamic RAMs. Familiarity with the Motorola 680x0 processors will also be quite useful.

While knowledge of Zorro II PIC design will also be useful, such experience mainly applies to the AUTOCONFIG sections of a PIC design. The signals and design problems for the Zorro III bus are substantially different than for Zorro II. Zorro III PICs are expected to run considerably faster than those for Zorro II, leading the circuit designer to faster TTL logic

families and more use of fast PAL devices. The additional speeds coupled with 32-bit buses will also lead the circuit board designer to multi-layer boards and more critical routing problems. While the Zorro II bus and most Zorro II designs are mainly synchronous, the Zorro III bus is asynchronous. Zorro III designs will typically be either fully asynchronous or self-clocked synchronous with proper attention to stable synchronization with the bus.

1.2 A Few Words About AUTOCONFIG

If past history is any indication, the first thing to mention about Zorro III PIC design is AUTOCONFIG, the Amiga mechanism for linking hardware plug-ins with software such that configuration jumpers for addresses are unnecessary, and device driver installation is trivial to even a novice user. And the first thing to say to a hardware designer about AUTOCONFIG is *Don't Panic*. More than any other issue, the AUTOCONFIG system seems to have confused Zorro II PIC designers. But there's absolutely nothing to fear about AUTOCONFIG; it is a very simple concept and very simple to implement as an integral part of any PIC's design.

The concept of configuration hasn't changed for Zorro III, and the implementation is very much the same as for the Zorro II bus. Extensions have been provided for a few Zorro III advanced features, and a few extra things were added to the specification to make the design of a 32 bit PIC as easy as possible. Other than that, if you know Zorro II configuration, you'll pick up Zorro III configuration almost instantly. Chapter 2 walks through the creation of an AUTOCONFIG circuit for Zorro III and discusses the basic logic likely to be in place on any Zorro III card.

1.3 Design Example Goals

The goal of this example is to design a memory card for the Zorro III bus. While A3000 users won't be running out of motherboard memory (up to 18 Megabytes) quite as fast as A2000 users did, there's already an emerging need for massive memory in Amiga computers. This RAM card meets the following goals:

- Provides a fully asynchronous design example
- Uses the same ZIP memories as the A3000
- Supports up to 8 Megabytes using 256K x 4 DRAMs, up to 32 Megabytes with 1M x 4 DRAMs.
- Hopefully functions as a relatively clear design example

And, of course, this is a fully functional design tested to the best of our ability at the time of this writing.

CHAPTER 2

AUTOCONFIG™ LOGIC DESIGN

"Logic is in the eye of the logician."

-Gloria Steinem

Every PIC design has a few things in common, most noticeably an AUTOCONFIG circuit. While such logic can pretty much be created by rote, an optimal design always will incorporate the AUTOCONFIG and other Zorro III bus logic naturally into the main design. While this chapter concentrates on the AUTOCONFIG logic, it will cover all of the standard logic elements of any Zorro III design in a sensible order.

Throughout this and the following chapters, references to the schematic pages in Appendix 2 will be. Page one of the schematics is found on page A-13 of this document, and there are six schematic pages. To make things simpler, these will be referred to as S-1 through S-6.

2.1 Bus Buffers

Just like with Zorro II, all Zorro III designs require a number of buffers on the bus logic signals. No PIC may load any bus signal with more than two F-series equivalent gates, and of course outputs from the PIC must be able to drive the bus properly. Any unbuffered signal used by a PIC must be used close to the bus connector; if a signal trace is longer than a few inches, it must be buffered. In addition, due to the dynamic nature of the high-order Zorro III address lines, some or all of these address lines must be latched for the duration of the bus cycle.

The buffering/latching arrangement is shown on S-1. Since this is a slave-only board, address lines are input-only. Addresses A₃₁-A₈ are transparently latched by 74F373 parts, the latch taking place when /FCS is negated. The transparent latching allows the address comparator to take advantage of the bus's address setup time, important for matching to the board's assigned address as quickly as possible. The circuitry shown here is the most straightforward, but in operation, only A₂₄-A₂ are actually used once the board select is determined. Thus, a fast enough comparator circuit can latch an address match rather than the high-order addresses if it saves on circuit complexity. Since the low order addresses A₇-A₂ are static, they are simply buffered coming into the RAM board. The extra buffers in that package are used in this design to buffer /FCS and READ, two lines used in several places in this design.

Data buffering is quite simple; D₃₁-D₀ are buffered with bidirectional bus buffers. The data direction and buffer enable signals are quite simple. The buffers point out toward the bus for read cycles when the PIC is selected (/SLAVE asserted), in at all other times; this function is contained in the U200 PAL. The output enable is asserted when the PIC is selected, the DOE signal is asserted, and there's no bus error; this function is contained in the U201 PAL. Because the data bus tristates, I use centering resistors to keep it quiet when it's not being driven. If this design had been supporting Zorro II as well as Zorro III, an additional two data buffers and much more complicated buffering logic, based on the SENSEZ3 line, would be required.

Reg	Bit	Val	Description
00	7,6	10	This indicates a Zorro III card.
	5	1	The OS will link this as free memory.
	4	0	No autoboot/diagnostic ROM.
	3	0	Only one logical PIC here.
	2-0	001	Using the extended size feature, this is a 32 megabyte board.
04	7-0	01010111	Commodore Product \$53.
08	7	1	Hint to the OS that this is memory, not I/O
	6	0	This board can be shut up.
	5	1	Extended sizing being used here.
	4	1	This must be 1, for 1.3 compatibility.
	3-0	0001	Let the OS calculate the logical size of the memory.
0C	7-0	00000000	Reserved.
10	7-0	00000010	Manufacturer's number, high byte.
14	7-0	00000010	Manufacturer's number, low byte. Since this one is a Commodore board, it uses the Commodore number.
18-3C	7-0	00000000	All of these are zeroed. This board does not contain a board serial number or boot/diagnostic ROM.
40	7-0	N/A	Reserved
44	15-0	CFGADDR	This board uses the Zorro III configuration block. It accepts the configuration address as a single write.
48	7-0	N/A	Configuration is completely handled with register 44.
4C	7-0	N/A	A write of any value will cause the board to shut up.
50-7C	7-0	N/A	All remaining registers are reserved.

Table 2-1: Logical AUTOCONFIG Registers

2.2 The AUTOCONFIG ROM

The complete AUTOCONFIG ROM is implemented in PAL U200, shown on schematic page S-2. The design of an AUTOCONFIG ROM is usually very simple, but it does require a complete understanding of how the board is to be used by the system before it can be done. Also, a Zorro III configuration ROM is similar to a Zorro II configuration ROM, with just a few more options available, once the translation for the configuration space chosen is applied.

First of all, the board must be described. Obviously, this is a Zorro III memory board, and since it's my design, it's also from Commodore. On top of that, it can be expanded up to 32 megabytes, and it can also be "shut up" if necessary. That's pretty much the specification, now it has to be translated into Zorro III ROM registers. *The Zorro III Bus Specification* describes these entries starting on page 8-1. The logical register assignments are illustrated in *Table 2-1*. The table actually lists all of the configuration registers on the board (registers 40-7C are reserved as write registers, not read registers, but they're mentioned here anyway).

The next step in the design process is to convert these bit assignments to actual logic. As mentioned before, the configuration ROM is implemented as part of the U200 PAL. By design, configuration ROMs fit nicely in a PAL in most cases. The Zorro II and Zorro III specifications call for all read registers other than register 00 to be inverted in their physical implementation. Since most bits are logically "0", they'll be physically "1", and "1" is the default output state of a standard PAL. Also taking into account that each logical register is actually made up of two

Address	D31	D30	D29	D28
00	1	0	1	0
02	0	0	0	1
04	0	1	1	0
06	1	1	0	1
08	0	1	0	0
0A	1	1	1	0
12	1	1	0	1
16	1	1	0	1
OTHERS	1	1	1	1

Table 2-2: Physical ROM Registers

physical registers, both of which assert data only on the D31-D28 nybble, the physical register mapping for all read registers is shown in *Table 2-2*. The actual PAL equations for this are on page A-3. These are simply a set of equations, one for each data line, that take into account each "0" in the above table, and are active only when the board is selected and not yet configured.

While it makes no difference to the equations for our ROM registers, it is a good idea to point out here the differences in addressing these read registers. Zorro II boards must respond to the configuration space \$00E8xxxx, and all registers are mapped on word boundaries. Zorro III boards can respond to the \$00E8xxxx address as a 16-bit Zorro II device as well, but many designs, including this one, will choose instead to respond to the Zorro III configuration space at \$FF00xxxx. A board responds to this address as a 32-bit device, and it actually need only decode the high-order eight bits of this address; both of these facts can save considerably on the amount of configuration logic necessary for some designs. In both configurations, the first nybble of each register pair is at the offset from base address given by that register number. In the Zorro II space, the second nybble is in the next logical word -- the register number plus two. Zorro III instead maps the second register of the pair at \$100 plus the register number. This may

sound like the two will be quite different in implementation, but as the example PAL U200 illustrates, if I map A₈ as A₁ in the equations, all ROM equations will be written the same for either configuration space. Using this feature and a multiplex of A₈ and A₁ based on the SENSEZ3 signal can help simplify the design of a card that adjusts to both Zorro II and Zorro III buses.

2.3 The AUTOCONFIG Registers

This design supports two writable configuration registers, the 16-bit configuration address register 44 and the shutup register 4C. Recall that configuration address registers are written in a pattern that allows the designer to choose nybble- or byte-wide configuration latches for Zorro II configuration space or byte- or word-wide configuration latches in Zorro III configuration space. Since Zorro II space is only sixteen bits wide and writes must line up consistently, this design would have to latch configuration address bits A₃₁-A₂₄ on a write to register 44, followed by configuration address bits A₂₃-A₁₆ on a write to register 48. Even though a large board such as this never needs to look at A₂₃-A₁₆ for its configuration address (Zorro III PICs always live at their natural boundaries), a board configured in Zorro II configuration space isn't configured until a write to register 48. Since this board instead responds to Zorro III configuration space, the entire sixteen bit configuration address can be written at once with a write to register 44, and that is also the signal indicating that configuration of the board is complete.

The register logic starts with the same PAL, U200, as used for our ROM logic. This PAL has the important low-order addresses going to it, so it's a natural for this. In this design, there are two signals created for register support in PAL U200. The first of these is a signal called /PRECON, for pre-configuration. The board isn't fully configured until the end of the Zorro III cycle that writes either register 44 or register 4C; /PRECON is asserted during this last write cycle as soon as data is valid on the bus, and it stays latched until the next reset. The other signal in U200 that's of immediate importance is the CFGLT signal. This line is responsible for latching the configuration address on the bus if this final write is a configuration and not a "shut up" request. This is an active high signal in an inverted-output PAL, so the equation can't be very complicated. This line is asserted when the board is selected, /PRECON is asserted, and A₃ is low, which is true just after /PRECON is asserted for a write to 44. Like the /PRECON line, CFGLT latches until the next reset. The remainder of the register logic is elsewhere.

The rest of the configuration control logic is in PAL U201, which creates both the /CFGOUT and /SLAVE signals, two signals that must be driven out to the backplane. The /CFGOUT signal is pretty simple. Normally, it is asserted at the end of a cycle in which /PRECON and /CFGIN are asserted, and latched asserted as long as PRECON also stays asserted. It also gets asserted if /CFGIN is asserted along with the SENSEZ3 signal negated. This latter condition indicates that the board has been placed in a Zorro II backplane. This board can't support Zorro II configuration, so it automatically "shuts up", an action required by the Zorro III specification. Note that the SENSEZ3 signal is called /Z2SHUNT in the PAL equations on page A-5.

The next basic piece of the configuration logic is the configuration latch, which in this case is the 74F374 at U202. This edge-triggered latch is triggered by the rising edge of CFGLT, which is asserted when the board's configuration address is written and data is valid on the data bus. At the end of the configuration address cycle, /CFGOUT is asserted, the address as latched is now fed into the /SLAVE generation address comparator, and the board is fully configured in hardware. Since this is an autosized memory board, system software generally will calculate its size and link it into the free memory pool before the next board is configured, though this operation can of course change as the configuration software changes.

2.4 The SLAVE Logic

Naturally, this brings up the question of how the /SLAVE logic is implemented. Every Zorro II or Zorro III board must assert its private /SLAVE line when it is responding to a bus address. In every case, two addresses must be supported; the configuration space address prior to configuration, and the software-assigned address after configuration. The method used in this example is quite similar to techniques used in many Zorro II designs, and is only slightly more complex.

The core of a /SLAVE circuit is always an address comparator of some kind. In every case, the bus address must be compared with the address to which the board responds. The main comparator in this circuit is the 74F521 at U203. It compares seven bits of possibly-latched bus address, A31-A25, with the corresponding bits on the configuration address latch. This comparison is called /MATCH on the schematics. Prior to configuration, the 74F374 is tri-stated, and the outputs going to the comparator are all pulled high, getting the card well on the way to responding to the \$FF00xxxx configuration space.

The twist in this design is that there is a bit more to this comparison than just a simple comparator can handle. First of all, the board needs to look at a full eight bits of the \$FF00xxxx address to properly respond during configuration, but only seven bits of address once the board is configured as a 32-megabyte board. This PAL U201 helps out by requiring A24 to be high for a /SLAVE response prior to configuration. Zorro III memory cards must monitor the function codes FC0-FC2. PICs must only respond to a valid User or Supervisor mode Code or Data space access; such accesses are given as the exclusive-or of FC0 with FC1. The /SLAVE signal is always qualified with the Zorro III full cycle strobe /FCS, and it can occur in only two cases. In the first case, a qualified match occurs, the board is unconfigured, and /CFGIN is asserted. In the latter case, a qualified match occurs, the board is configured, and CFGLT is asserted. As previously mentioned, if the board is configured but CFGLT is negated, the board has been "shut up" rather than configured.

And that is all there is to the basic configuration logic. As demonstrated with U201, it is usually quite reasonable to incorporate this logic in with other board logic, where it'll fit the most efficiently. AUTOCONFIG logic is intended to make it easy on the designer as well as the user; it's not supposed to scare anyone.

CHAPTER 3

MEMORY SYSTEM DESIGN

"I like them big and stupid."

-Julie Brown

This chapter discusses the actual DRAM logic design for this project. The information here is going to be far less useful for the designer trying to learn about Zorro III designs, since this is the part of the board design which is very specific to the task at hand, a DRAM board. However, there may be some ideas of a more general applicability. If nothing else, it shows that a fully asynchronous DRAM design can be done rather simply with a little planning.

3.1 DRAM Refresh

The most complex part of any hand-made DRAM circuitry is very likely to be the refresh circuitry. Without refresh, DRAM would look pretty much like static memory with a multiplexed address bus (and the folks at TI and National Semiconductor would be selling quite a few less DRAM controllers). While there's nothing wrong with off-the-shelf DRAM controllers, it's really not very difficult to "roll your own".

3.1.1 Refresh Arbitration

If you believe that DRAM boards are difficult to design, and that refresh is the most difficult part of such a design, then you must believe that refresh arbitration is the most difficult part of the refresh logic. So I'll discuss that part first, and the rest of the circuitry in this chapter will then be simpler.

In this design the refresh arbiter is incorporated with the /SLAVE generator. Refresh arbitration takes place in U201 and is by the nature of the Zorro III bus necessarily asynchronous. A refresh request can come in at any time, and must be serviced as soon as possible without interrupting a cycle. There are three refresh cases: a request outside of a cycle, a request during a cycle to this card, and a request during a cycle to another card. These cases are illustrated in *Figure 3-1*. The problem with any asynchronous refresh arbiter is that it's impossible to determine at a single point if a cycle is starting or not. This can be thought of as a potential race condition between the refresh request and the start-of-cycle. So the solution is to create two sampling points, one to give the go-ahead for a refresh cycle, the other to give the go-ahead for a memory cycle.

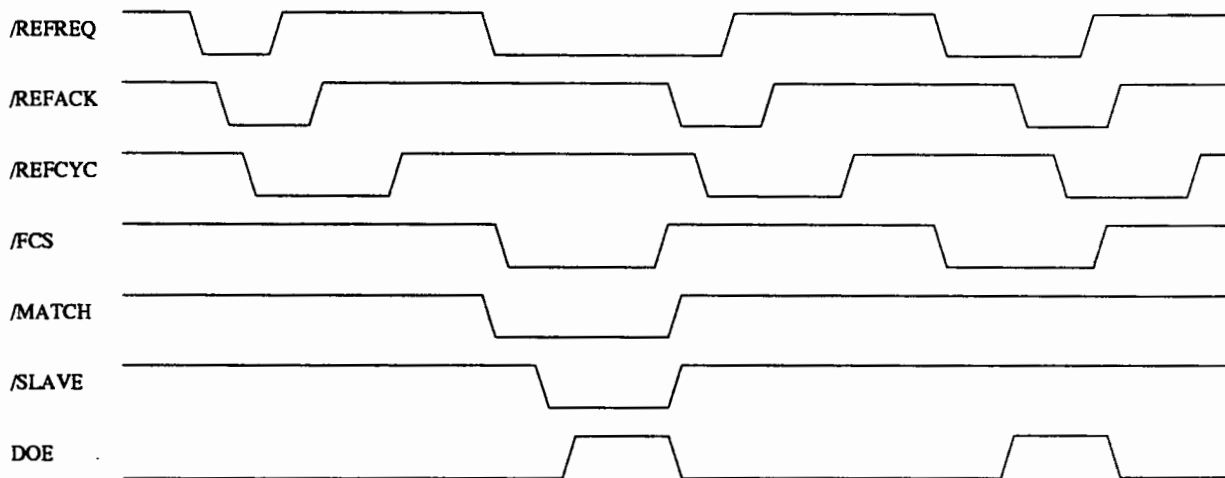


Figure 3-1: Refresh Arbitration

For the latter, you can use the /SLAVE signal. Virtually everything that happens on a simple Zorro III slave card is gated with /SLAVE. So in order to safely arbitrate refresh, we generate a refresh acknowledge signal, /REFACK, which will always be asserted safely before or safely after /SLAVE. In order to get there before /SLAVE, the /REFACK line will not be asserted outside of a Full Cycle if the /MATCH line is asserted. Since /MATCH and /FCS must both be asserted in order to create /SLAVE, and /FCS always follows /SLAVE, the /REFACK line is guaranteed to get out of U201 prior to /SLAVE, should the refresh request come in just before the PIC is selected. But once a board is selected on the bus, there's no reason to hold off refresh if it's a different board being selected, so /REFACK can be asserted during the data time of some other card's full bus cycle. In either case, the refresh acknowledge is latched as long as refresh request is held.

3.1.2 Refresh Counter

A simple refresh counter is implemented in PAL U306. Although the board supports both 256K x 4 or 1 Meg x 4 DRAM, the actual per-row refresh time is the same; the former part requires a 512 row refresh in 8ms, the latter a 1024 row refresh in 16ms. This amounts to one refresh request every 15,625ns. However, to build in support for burst mode with page-mode or

static column DRAM, we use the $TRAS_{MAX}$ time here, which is 10,000ns. The PAL counter actually counts 140ns clocks, so a count of 71 clocks will get us up to 9,940ns, close to the desired 10,000ns. If burst mode support weren't considered here, a count of 111 clocks could be used in the counter.

The counting is quite simple; the counter goes from zero to its terminal count, then asserts the $/REFREQ$ signal. It then holds onto the $/REFREQ$ signal until a refresh cycle is under way, as indicated by $/REFCYC$. The $/REFCYC$ line will reset the counter for the duration of the refresh cycle. The process starts over once the refresh cycle is complete.

The clocked counter is used here simply because it's very easy to understand and, being fully digital, always works the same way. It could have been a simple one-shot or 555 timer circuit, as long as component tolerances don't allow the timer to drop below the required refresh frequency. You may recall reading of the evils of such timers in DRAM hint books. While they

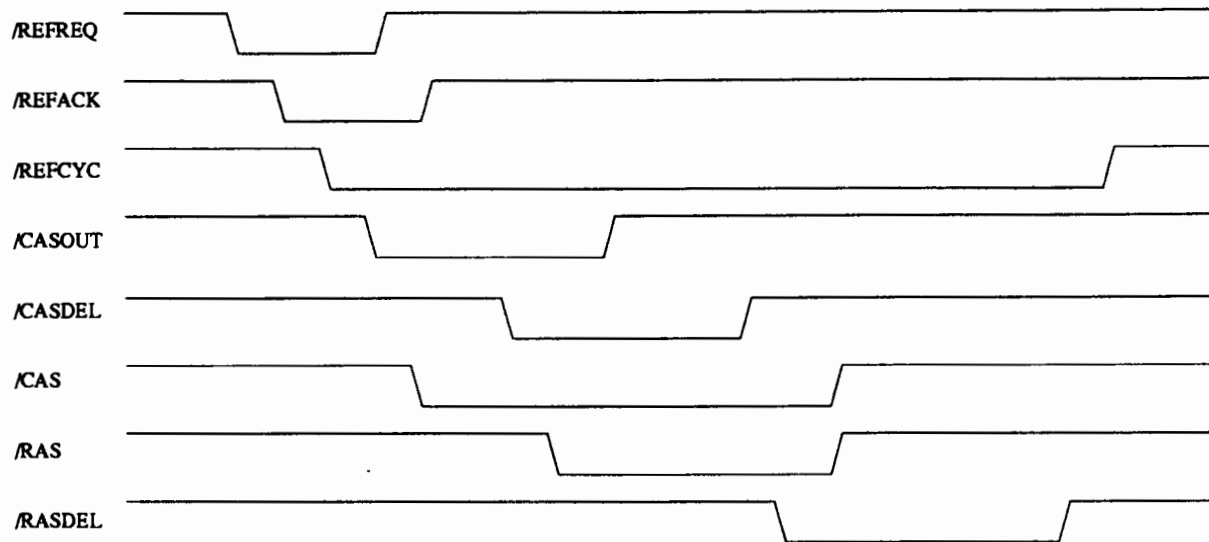


Figure 3-2: Refresh Cycle

aren't optimal, due to the aforementioned component tolerance problems, that's not why you were warned off. The main reason for avoiding such timers in most DRAM designs is the problem you're likely to have with an asynchronous refresh request. Since we have already solved the problem of the asynchronous refresh request here, no asynchronous approach is inherently evil to this design.

3.1.3 Refresh Cycle

The actual refresh cycle, illustrated in *Figure 3-2*, is a CAS-before-RAS refresh, and all memory on board is refreshed at the same time. As soon as a refresh cycle is active ($/REFCYC$ asserted), PAL U300 will assert the $/REFCAS$ line. $/REFCAS$ will in turn cause the CAS control PAL, U304, to drive all eight CAS lines. An active $/CASDEL$ or $/RASDEL$ will hold off

the assertion of /REFCAS, thus ensuring RAS precharge (TRP) in case the refresh is immediately following a memory cycle. The /REFCAS line is latched by /MUX until /RASEN comes along, so that it's no longer dependent on /REFACK. The /REFACK line will be negated some time before the end of the CAS-RAS cycle; its main use here is to qualify the start of a refresh cycle. Once the /RASEN is asserted, /REFCAS is latched by the negated /RASDEL, as is /RASEN.

The /CASOUT line of U300 is also driven at the start of the refresh cycle. This of course comes back to U300 as the /CASDEL signal. The refresh /RASEN is driven as soon as /CASDEL is asserted, thereby separating refresh CAS and refresh RAS by roughly the CAS delay time. The /RASEN line drives the buffered /RAS lines to either bank of memory. Once asserted, /RASEN is held until /RASDEL wraps back in. The refresh cycle is held until /RASDEL once again is negated, thus ensuring TRP for the refresh cycle, in the event that this refresh is taking place right before a memory cycle.

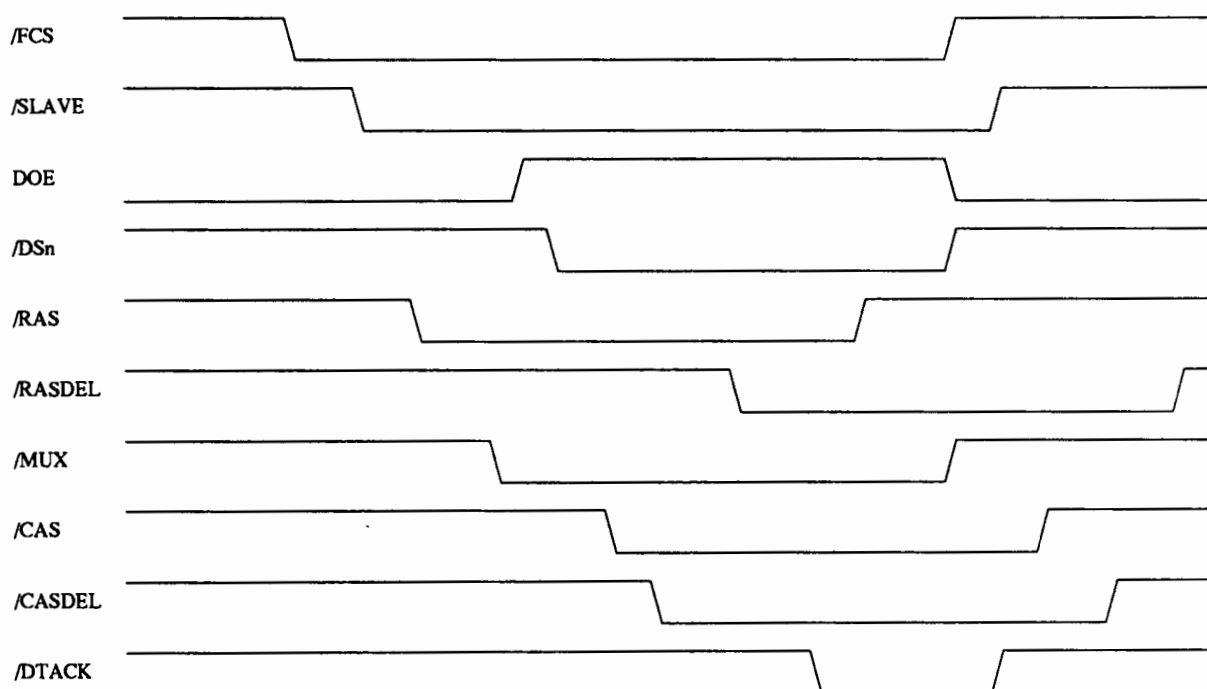


Figure 3-3: Memory Access

3.2 DRAM Access

The DRAM read/write access during a normal memory cycle uses mainly the same parts for RAS-CAS controlling, along with a few additional bits and pieces to control memory banking. The logic supports several speeds of DRAM, selection being made via jumpers on the tap delays used for RAS and CAS timing. Either the 256K x 4 or 1 Meg x 4 parts can be used, and a jumper is provided to allow the necessary banking modification for this. Finally, hooks are in place for burst-mode (Multiple Transfer Cycle) support of either page or static column DRAMs, but at the time of this writing, Zorro III burst mode is not yet implemented in the A3000 Bus Controller, so this feature can't yet be tested.

3.2.1 Memory Cycle

The basic memory cycle is started by U300 when /SLAVE is asserted and no refresh cycle is acknowledged or in progress. The cycle start will be held off until /RASDEL is negated, to ensure TRP after a refresh or a previous memory cycle. The assertion of /RASEN starts the cycle, and /RASEN is held at least until /DTACK is asserted. Dropping /RAS before cycle's end lets us get an early start on RAS precharge, and by /DTACK time the appropriate /CASXN are certain to have fallen, assuring that data will be held through the cycle's end. Since /RASEN creates /MUX, however, this optimization can't be used for SCRAM parts, since that could result in a column address change before cycle's end (SCRAMs don't latch the column address). The 100ns tap delay U301 sets the RAS delay, and J300 provides taps for 100ns, 80ns, and 60ns DRAM. The /RASEN line is buffered, as previously mentioned, by two gates from the 74F244 at U303, one creating /RASL for the lower bank of 32 memories, the other creating /RASH for the upper bank of 32 memories. U303 also buffers the first tap from U301, which becomes /MUX, the line used for multiplexing the DRAM addresses.

The U300 PAL also creates the enable for CAS, the /CASEN line. This is based on /RASEN, DOE, and /MUX asserted, and it's held through the end of the cycle, until /DTACK is negated. The /CASEN line qualifies CAS, but it doesn't necessarily start CAS for a full cycle; further consideration of CAS generation is done elsewhere. There are hooks in U300 to change the operation of /CASEN in the case of Multiple Transfer Cycles and either page-mode or static column DRAM. There's logic intended to support this in the PAL equations, but it has not yet been tested.

Most of the CAS generation is handled in U304, the CAS generation PAL. The CAS strobes are used to select between two banks of DRAM, and to select the appropriate bytes to access during write cycles; this is covered in detail in the next section. Other than qualifying by bank and byte, the CAS generation PAL qualifies all CAS with READ. During read cycles, all four bytes in the accessed memory bank are activated, in order to support caching of this memory. Write cycles, on the other hand, are qualified with the appropriate data strobe, to assure that data is valid before a write-cycle CAS latches write data. All CAS strobes are of course qualified by /CASEN. They're also all qualified with /CADDR, which is a strobe that assures column address setup time to CAS. This is just the 60ns tap from the RAS timing tap delay. The 40ns tap would just about make it, but leaves absolutely no margin. Since column access is rarely the limiting factor, the 60ns tap is used, for a 30ns worst case /MUX to /CADDR delay, assuming a 5% per-tap tolerance on the tap delay.

3.2.2 Bank Selection

The refresh cycle's CAS-before-RAS logic, along with the fact that the whole board is refreshed at once, keeps things pretty simple when refresh is taking place. A normal memory cycle, however, must take into account the memory devices that actually need to be addressed. This discussion is concentrating mainly on the 256K x 4 devices, but the same principles apply to the 1M x 4 devices as well.

The basic memory unit is a 4-bit DRAM, and thus two devices are necessary to form a byte, the basic unit of interest to the Zorro III bus. This makes the smallest chunk of 32 bit memory a one megabyte chunk. So for the total of eight megabytes, we'll have eight 1-megabyte memory banks. We want to keep RAS common among all DRAM, so it can't be used to control banking at all. The best thing to do is divide and conquer, and that's just what we do; find something to select between these various natural divisions.

As mentioned previously, the /CAS strobes are used to select individual bytes within a one megabyte bank of memory. This is a very natural use of /CAS, since it's not needed until late in the memory cycle, and the data strobe lines and write data aren't valid until later in the cycle either. The CAS PAL could easily generate a /CAS₀-/CAS₃, based directly on corresponding data strobes /DS₀-/DS₃. However, there are twice the number of output lines on this PAL device as needed for four /CAS lines, and we're still looking for a banking mechanism. With the addition of the MEG4 signal for memory sizing and the address lines A₂₂ and A₂₄, the PAL comes to drive eight total /CAS lines, controlling not only byte enables but the most significant RAM bank. For 256K × 4 parts, A₂₂ chooses between two 4-megabyte banks. For 1M × 4 parts A₂₄ chooses between two 16-megabyte banks.

Within the 4-megabyte banks, another banking control is used. In this case, most of the work is done by the 74F138 decoder at U305. This device creates a read enable for one of four device during a read, or a write enable for one of four devices during a write. The selection of device is controlled by the BK₀ and BK₁ lines from U300. BK₀ and BK₁ are simply A₂₀ and A₂₁ for 256K × 4 support, or A₂₂ and A₂₃ for 1M × 4 support. That's all there is to bank selection. Zorro III autosizing requires board memory to be added from the lowest to the highest address on-board, but there are no hardware requirements for this.

3.2.3 Address Multiplexing

There's nothing really complicated about the address multiplexing on this card, but it should be explained. All of the multiplexing is done with 74F258 multiplexers, and all of them are multiplexed by the /MUX signal. The first four or sixteen megabytes of memory is driven by /MAL₀-/MAL₉, the second by /MAH₀-/MAH₉, but the multiplexing scheme is identical for both banks. When /MUX is high, the row addresses /MA₀-/MA₉ are set to the inverted A₁₀-A₁₇, A₁₉, and A₂₁, respectively. For /MUX low, the column addresses /MA₀-/MA₉ are set to the inverted A₂-A₉, A₁₈, and A₂₀, respectively. This organization may seem strange, but it makes A₂-A₇ (the Multiple Transfer static addresses), the low-order column addresses, so that Multiple Transfer Cycles can be supported via fast page or static column DRAM. This banking scheme also makes /MA₉, which is used only by 1M × 4 DRAM, a no-op for 256K × 4 DRAM, since BK₀-BK₁ look at A₂₀ and A₂₁.

CHAPTER 4

GOING FURTHER

"There is more to life than increasing its speed."

-Mahatma Gandhi

In the general case, you can always do better. When specifics get involved, though, you may not always want to. In the specific case of this design example, you can certainly do a bit better. And if you want to make this example into a real product at some point, you should do better (thanks to this article, anyone can do at least this well just by copying).

Currently, with the Revision G Buster chip in a 25MHz A3000, this design with 80ns DRAM is running at just over half the speed of A3000 local bus memory. But part of that is the current Zorro III implementation -- this same configuration is running only about 15% slower than our prototype 50ns SRAM board! You can fully expect the Level 2 Buster chip to improve cycle times considerably, as well as supporting the faster MultipleTransfer Cycles. So, as I said, you can always do better.

4.1 Designed-In Enhancements

While not quite in the "quick and dirty" category, this example went from start to final working version in about five working days. Most of the careful design work was spent on getting the AUTOCONFIG logic correct and understandable, since that's the most likely part of the design to be replicated in other Zorro III PICs. The actual DRAM part of it was designed, above all else, to work right the first time, since there really wasn't any time to revise the board. Because I felt that presenting a design example at a Developer's Conference without a working

sample in hand would certainly be a cause for developers worry about the design's quality. So this card was designed to work, above all other concerns.

4.1.1 The Experimenter's Board?

As it turns out, the original concept for the DRAM memory cycle worked fine, but the refresh logic has a rather serious flaw that hadn't been considered originally. When the design was created, the /REFACK signal was seen as the refresh control that stays valid for the entire refresh cycle, while the /REFCYC signal, then called /REFHOLD, was an end-of-cycle signal used to control the RAS precharge delay. That didn't work, and fortunately, the current mechanism could be created by changing the PAL equations, so the board was working a day after it was built up without a single cut or jumper.

However, the original memory cycle left a bit to be desired. Initially, the CAS enable didn't go out until the full RAS time had been met (eg, /RASDEL is asserted). This worked, but made CAS quite a bit later than it could have been. With a single extra wire, the CAS PAL was modified to hold off CAS until column addresses became valid. This allowed the memory timing PAL to enable CAS as soon as possible, and resulted in a 15% speedup.

The point here is that the design, as presented, isn't completely fixed. There are a considerable number of things one could do to change the memory cycle by playing around with PALs. It's conceivable that even without any additional PCB modifications, the memory cycle efficiency could be enhanced.

4.1.2 Multiple Cycle Transfer Support

One enhancement that's definitely supported, though untested, is the Zorro III Multiple Transfer Cycle. PAL U201, when enabled by J200, will request Multiple Transfer Cycles, and drive the /BURST line if the bus master acknowledges this with /MTCR. The memory controller PAL U300 attempts to create proper CAS cycles for a burst transfer, modified by the J303 jumper for fast page or static column mode DRAM. And, as previously mentioned, the address multiplexing and refresh timeout are designed to support this burst mode as well. Hopefully this logic would work with a Level 2 Buster that can handle bursts, but it hasn't been tested at the time of this writing.

4.2 Modification Ideas

Opening up the design to a few PCB modifications can make things much more interesting. Of course, the ultimate modification might be to throw out the complete DRAM logic here and simply go to an off-the-shelf DRAM controller. While there's nothing wrong with that approach, and modern DRAM controllers even have an asynchronous operating mode that would work very nicely with Zorro III, there is still some performance that can be squeezed from this basic design. Most of these might have been incorporated with an extra day or so worth of design time.

4.2.1 Tighter RAM Cycles

The entire memory cycle run here is a bit less than optimal. Part of the problem is that the memory timing and CAS control PALs don't always have the same idea of when CAS should start. If the controller has a very good idea of when data is going to become valid, whether driven by TRAC or TCAS, the /DTACK line can be driven optimally. And, of course, the cycle can be fully TRAC driven, which is usually going to be the fastest possible cycle.

Another less than optimal feature of the design is the TRP assurance logic. In order to manage TRP between a cycle immediately following refresh or refresh immediately following a cycle, all new cycles are held off until /RASDEL is negated. This works just fine, but the time between /RAS negated and /RASDEL negated is very close to the TRAS time. For all standard DRAMs, the TRP time is less, sometimes much less, than the required time for TRAS. The CAS precharge time is never a problem for full cycle to full cycle operation, and unlikely to be a problem for Multiple Transfer Cycles.

The built-in support for Multiple Transfer Cycles can also be improved. The main problem for such burst cycles that doesn't crop up elsewhere is the TRAS_{MAX} time of most DRAMs in burst or static column modes. This board makes sure that a burst transfer can't exceed this limit by setting the refresh time to something just under TRAS_{MAX}. When refresh comes along, it causes /MTACK to be negated at the appropriate subcycle boundary, thus making the full cycle terminate so that refresh can take place. This has two shortcomings. First of all, it makes refresh related slowdowns over 50% more likely than necessary. Additionally, the start of the burst cycle isn't synchronized with the refresh counter, so a burst can be interrupted by refresh long before necessary. Ideally, separate counters could be added for burst and refresh timeouts. Alternately, the refresh counter could be modified to change its count based on whether or not a burst cycle is under way.

4.2.2 Read/Write Optimizations

A basic principle of Zorro III slave optimization is that read and write cycles can benefit from different treatments. In this example, for instance, CAS can be driven before the DOE signal is received for read cycles, as long as column addresses are valid. If data can be valid on the card's data bus prior to DOE, then the cycle can be acknowledged only one buffer enable time after DOE is received. For READ sensing in the DRAM timing PAL (U300), the addresses used for the DRAM banking logic can easily be moved into another device, freeing up about seven pins.

Write optimizations would take a bit more logic, but they are possible. The best write enhancement would be data bus latches. By replacing 74F245 buffers U104-U107 with some 74F646 bidirectional latching buffers, and associated control logic, writes can be made very fast. The falling edge of the /DS_N lines can latch data to the board and effect an immediate /DTACK, thereby possibly saving some of the TRAS and TCAS time. In fact, this could also help reads, since a latched data bus would allow the DRAMs to shut off as soon as data's latched, rather than at the end of the Zorro III cycle.

4.2.3 Standard DRAM Tricks

As with any DRAM design, the standard DRAM tricks apply here. With a bit of logic duplication, doubling up on the RAS-CAS and refresh logic, memory bank interleaving can be used to hide the RAS precharge time in most cases. Multiple Transfer Cycles can be thought of as an automatic page detect, so conventional page mode or static column optimizations may not be all that useful. Then again, the Zorro III page is only 256 bytes, so perhaps a larger page could be of some help. Nybble mode memories won't really be of much use; although any burst cycle resulting from 68030 burst mode will be nybble compatible, there's no guarantee of linear addressing within a Zorro III burst cycle.

Always keep in mind the future. The Zorro III bus implementation that's currently on the A3000, as mentioned before, is already slated to improve. In the future memory will go faster on the bus than it does now, even if motherboard clocks don't go beyond 25MHz. And we expect future Zorro III machines will be running a faster Zorro III bus, going beyond what's possible in an A3000 even tomorrow.

CHAPTER 5

ADDITIONAL ZORRO III ADVICE

"Cute rots the intellect."

-Garfield

Going beyond this specific example just a bit, there are a few good things to think about when working on any Zorro III design. A large portion of this is just plain good design sense. Those without much design sense or experience should read this chapter twice, and probably learn more about the first two points from some outside materials.

5.1 Watch Those Synchronizations

The foremost thing to be concerned about when designing for Zorro III is the fact the bus is running asynchronously. Some simple designs will not find this to be any problem. Obviously a simply I/O chip with a 100ns access time can be timed with a delay line, keeping things very simple. At the other end of the spectrum of complexity, clever clocked VLSI chips often internally synchronize things, much the way the 680x0 processors handle their "asynchronous" inputs.

If, however, you're doing your own TTL level design, such as this one, be very careful. Fully asynchronous circuits can be very tricky to do correctly, missing a strobe by a nanosecond or so can be fatal, and it may only happen every so often. The best bet is to use overlapping signals and feedback to create new signals, and *never* count on delays through PALs or TTL to provide repeatable delays. Tap delays, while not perfect, are reasonably accurate, and can be used to design reliable circuits.

Synchronous design is usually easier, and therefore more reliable for the average designer to create. The problem here is coupling the synchronous design to the Zorro III bus. Such a design will have its own clock, but that clock can't reliably sample any Zorro III signal on a single edge. Double clocking any important Zorro III inputs with high quality flip-flops that go to clocked logic is a necessity. The problem you'll have with single clocking, metastability, won't always be immediately noticed, but it's going to be there. Better to avoid it from the start.

5.2 Design for Speed

Zorro III cards currently run around four times faster than Zorro II cards, and the limit, at least in theory, is over ten times faster. That should be a good indication that Zorro III designs are more sensitive to problems than Zorro II cards. To further aggravate the situation, you may not see any problems until faster Zorro III bus masters come along. So proper design practices are your best option. There are three main design problems that typically come up.

The speed of the design is one problem area, though it's not that much of a problem if you're up-to-date on the logic of the 1990s. While FCT and F series TTL are good for buffers and small logic functions, most fast designs these days rely heavily on programmable logic, mainly PALs. A single level of PAL logic can replace several levels of TTL, and they're always pushing PAL speeds just a little bit more. Larger PLDs and gate arrays (programmable or custom) are always handy for complex circuits, providing they're fast enough.

Noise problems are partially a result of the higher speeds involved. Eliminating such problems is achieved via a combination of circuit design and PCB layout. For noise reducing design, you need bypass capacitors of various sizes in the appropriate places. Every TTL part should have a small capacitor; we generally use something in the 0.1 μ F-0.22 μ F range. For DRAM or other surging parts, we use 0.33 μ F or greater. It's also a good idea to have a high frequency bypass, maybe 0.01 μ F or so, and a couple of larger capacitors, something in the 10 μ F-100 μ F range, randomly distributed around the design. More noise reduction can be achieved with good signal termination. Small value series termination resistors, something in the 22 Ω -68 Ω range works well; the values must often be tuned to the design. Tri-statable buses often benefit from some kind of parallel termination; pullups, pulldowns, or centering resistors depending on the design.

The other half of the noise problem is solved in PCB layout. Zorro III boards are almost certainly all multi-layer boards. Trace lengths are to be kept as short as possible, especially those on the bus side of a card; it's extremely important to minimize the noise that a card introduces to the bus. Fast and noisy signals, such as clock lines or fast control signals, should generally be given priority when routed. Component placement is also a very important job; the lengths of interconnects is directly affected by this planning. If the circuit designer isn't doing the board layout personally, he/she should develop a good working relationship with the PCB designer. Any work done on keeping the design quiet will very likely be time well spent; it's likely to help out in reliability, operation with other boards in the system, and government noise certifications such as FCC or FTZ.

5.3 Follow the Specifications

Let's say it once again! Any current Zorro III bus implementation is likely to be far more relaxed than the bus specification. That's going to eventually change. A proper design built today should work in tomorrow's 50MHz superAmiga, a substandard design could fail on an A3000 with the Enhanced Buster chip. Build in your long term viability at the design stage and save a great deal of potential future grief. You aren't going to get tested on your design for some time to come.

APPENDICES

"It ain't the meat, it's the motion"

-Southside Johnny

A.1 PAL Equations

The following section contains the complete PAL equations for the five PAL devices in the BIGRAM design. All the equations are in the CUPLTM format, but should be easily translated to any other format if required. This format uses the & character to represent AND, the # symbol to represent OR, the \$ symbol for XOR, and the ! symbol for negation. Standard outputs are indicated simply by name, registered outputs are indicated with the .D extension, and output enables are indicated with the .OE extension. The CUPLTM compiler minimizes equations where possible; should any equations here appear to be too large, rest assured that they will actually fit in the specified PAL.

A.1.1 Autoconfiguration Control PAL

This device is responsible for providing the AUTOCONFIG™ ROM, registers, and data buffer direction control. This is to be programmed into a 15ns 16L8 or equivalent device.

```

PARTNO      U200 ;
NAME        U200 ;
DATE        May 30, 1990 ;
REV         2;
DESIGNER    Dave Haynie ;
COMPANY     Commodore-Amiga ;
ASSEMBLY    BIGRAM ;
LOCATION     U200 ;

/*****
/*
/* Zorro III BIGRAM Configuration Control
/*
/* This device acts as configuration ROM and configuration
/* register controller.
/*
/*
*****/
/*
/* DEVICE DATA:
/*
/* Device:      16L8-15
/* Clock:      NONE
/* Unused:     NONE
/*
*****/

/* INPUTS: */

PIN 1  =      !SLAVE      ;      /* Board selected? */
PIN 2  =      !RST       ;      /* Board reset */
PIN 3  =      !DS3       ;      /* High order data strobe. */
PIN 4  =      READ       ;      /* Read cycle strobe */
PIN 5  =      A2         ;      /* Bus Addresses. */
PIN 6  =      A3         ;
PIN 7  =      A4         ;
PIN 8  =      A5         ;
PIN 9  =      A6         ;
PIN 11 =      A1         ;      /* This is really A8. */
PIN 16 =      !CFGOUT    ;      /* Board configured? */

/* OUTPUTS: */

PIN 19 =      D28        ;      /* Configuration data ROM nybble. */
PIN 12 =      D31        ;
PIN 13 =      D30        ;
PIN 14 =      D29        ;
PIN 15 =      DBDIR      ;      /* Data buffer direction. */

/* BIDIRECTIONALS: */

PIN 17 =      !PRECON    ;      /* Preconfiguration strobe. */
PIN 18 =      CFGLT      ;      /* Configuration address latch. */

/** INTERNAL TERMS: **/

/* Mapping A8 as A1 here makes the register pairs line up just
as they would under Zorro II configuration. */

field addr      = [A6..1];

/** OUTPUT TERMS: **/

/* The configuration ROM is created here. The logical ordering
of it is as follows:

REG      76543210

00      10100001    Zorro III, autolink, 32 megabytes
04      10010010    Product $53
08      10110001    Extended Memory board, supports
                   Shutup, autosized in software.
0C      00000000    Reserved
10      00000010    Manufacturer's code (C-A)
14      00000010
18-3C   00000000    Zeroed options/reserved.

The autoconfiguration specs call for every readable register
except for 0 to be inverted in the physical implementation.
So the resulting map is:

```

ADDR	D31	D30	D29	D28
00	1	0	1	0
02	0	0	0	1
04	0	1	1	0
06	1	1	0	1
08	0	1	0	0
0A	1	1	1	0
0C	1	1	1	1
0E	1	1	1	1
10	1	1	1	1
12	1	1	0	1
14	1	1	1	1
16	1	1	0	1
OTHERS	1	1	1	1

Only the Zero terms are explicitly entered here; anything not specifically driven low will be driven high.
*/

```
!D31      = addr:02
          # addr:04
          # addr:08;
```

```
!D30      = addr:00
          # addr:02;
```

```
!D29      = addr:02
          # addr:06
          # addr:08
          # addr:12
          # addr:16;
```

```
!D28      = addr:00
          # addr:04
          # addr:08
          # addr:0A;
```

```
[D31..28].OE = SLAVE & !CFGOUT & READ;
```

/* This signal is driven to indicate an address latch request. Note that the board uses 16 bit configuration write feature to configure all at once; this isn't available in the Zorro II configuration space. */

```
CFGLT     = SLAVE & PRECON & !A3
          # CFGLT & !RST;
```

/* If the board is told to shut up or configure, this line is asserted and held through reset. The logical SHUTUP line is PRECON & !CFGLT, once FCS is negated. */

```
PRECON     = SLAVE & DS3 & !READ & addr:4C
          # SLAVE & DS3 & !READ & addr:44
          # PRECON & !RST;
```

/* This controls the data buffer direction between the PIC's local bus and the expansion bus. */

```
DBDIR      = SLAVE & READ;
```

A.1.2 Board Control PAL

This device controls an assortment of board functions. It creates the /SLAVE, /CFGOUT, and /MTACK signals for Zorro III. It creates the data buffer enable for the bus buffers, and the burst-enable line used by the memory system. And it arbitrates DRAM refresh. This is programmed into a 10ns 20L8 or equivalent PAL.

```

PARTNO      U201 ;
NAME        U201 ;
DATE        May 30, 1990 ;
REV         3 ;
DESIGNER    Dave Haynie ;
COMPANY     Commodore-Amiga ;
ASSEMBLY    BIGRAM ;
LOCATION     U201 ;

/*****
/*
/* Zorro III BIGRAM Board Control
/*
/* This device controls the main features of the BIGRAM board.
/*
*****/
/*
/* DEVICE DATA:
/*
/* Device:      20L8-10
/* Clock:      NONE
/* Unused:     22(O)
/*
*****/

/* INPUTS: */

PIN 1  =      !MATCH      ;      /* Address match from comparator. */
PIN 2  =      CFGLT       ;      /* Configuration latch. */
PIN 3  =      !PRECON     ;      /* Board was configed or shutup. */
PIN 4  =      !FCS        ;      /* Full Cycle Strobe. */
PIN 5  =      !CFGIN      ;      /* Configuration chain in. */
PIN 6  =      FC0         ;      /* Function codes, don't ignore these! */
PIN 7  =      FC1         ;
PIN 8  =      !REFREQ     ;      /* Refresh request from refresh counter */
PIN 9  =      !Z2SHUNT    ;      /* Zorro II backplane bypass. */
PIN 10 =      DOE         ;      /* Data enable. */
PIN 11 =      !BERR       ;      /* Bus error, all off. */
PIN 13 =      !REFCYC     ;      /* We're in a refresh cycle. */
PIN 14 =      !BRENB      ;      /* Burst/Multiple transfer enable. */
PIN 20 =      !MTCR       ;      /* We're in a multiple cycle. */
PIN 23 =      A24         ;      /* Latched bus address 24. */

/* OUTPUTS: */

PIN 15 =      !DBOE       ;      /* Data buffer output enable. */

/* BIDIRECTIONALS: */

PIN 16 =      !CFGOUT     ;      /* Board is configured. */
PIN 17 =      !REFACK     ;      /* Refresh acknowledge. */
PIN 18 =      !MTACK      ;      /* Multiple transfer acknowledge. */
PIN 19 =      !SLAVE      ;      /* Board select. */
PIN 21 =      !BURST      ;      /* This is a burst cycle. */

/** INTERNAL TERMS: **/

/* The valid board address consists of a comparator match and a valid
memory space. The valid spaces are as follows:

SPACE      FC2 FC1 FC0
Reserved   0  0  0
User Data  0  0  1
User Program 0  1  0
Reserved   0  1  1
Reserved   1  0  0
Supervisor Data 1  0  1
Supervisor Program 1  1  0
CPU        1  1  1

This reduces to the equation used: FC0 XOR FC1. The external comparator
only looks at A31..A25, which is OK for normal operation (we're a 32
meg board), but bad for configuration. So if we're not yet configured,
A24 must be high for a select match.
*/

select      = MATCH & (FC0 $ FC1) & (CFGOUT # A24);

```

```

/* This indicates a normal board select; SLAVE starts the cycle, FCS
cuts it off quickly at the end. */

hit                = SLAVE & FCS;

/* OUTPUT TERMS: */

/* This output controls the data buffer enable pins. Data buffers
turn on when DOE is asserted and the board is selected, they
turn off as quickly after a cycle ends as possible. */

DBOE              = hit & DOE & !BERR;

/* This signal indicates that the board is configured. The board is
considered configured if actually configured, shut up, or placed
in a Zorro II backplane. It only responds if actually configured,
of course. This signal must only change at the end of a cycle, if
actually operating. */

CFGOUT            = PRECON & CFGIN & !FCS & !DOE
# PRECON & CFGOUT
# Z2SHUNT & CFGIN;

/* This is the refresh acknowledge cycle. When the a refresh request
comes in, and the coast is clear, this line is asserted to start
the refresh machine. Determining when the coast is clear, eg,
arbitrating refresh, is the trick to all hand-made DRAM controllers.
This one works pretty simply. The coast is clear when there's no
bus cycle happening, or when a bus cycle is happening but another
slave is responding. The trick is avoid races; FCS could be
changing just as REFREQ comes in. Therefore, the second half of
this arbiter is in the RAS cycle generation, which doesn't start
until REFACK is negated and SLAVE is asserted. */

REFACK            = REFREQ & !FCS & !MATCH
# REFREQ & FCS & !SLAVE & DOE
# REFACK & REFREQ;

/* The multiple cycle transfer acknowledge. If the jumper enables
them, and a refresh isn't already requested, we'll acknowledge
them. If a refresh request comes in, we'll negate MTACK after
the current cycle finishes, which will result in one more
burst cycle before the full cycle terminates and the refresh
can be acknowledged. I do it this way because I use the
refresh timer to handle the TRASMAX limitation of the DRAM as
well as handling refresh. */

MTACK             = hit & BRENB & !REFREQ
# hit & MTACK & !DOE
# hit & MTACK & MTCR;

MTACK.OE         = hit;

/* This is SLAVE, the board select line. Most board activity centers
around this line. If the board is selected and unconfigured,
always respond. Once configured, only respond if it's not shutup
or shunted. This line is held through the cycle's end. */

SLAVE            = select & FCS & CFGIN & !CFGOUT
# select & FCS & CFGLT & CFGOUT;

/* This indicates if the cycle is a burst cycle. The first cycle is
always a non-burst cycle. If, at the end of the first cycle,
MTCR and MTACK are asserted, all subsequent cycles are burst
until FCS is negated. */

BURST            = SLAVE & DOE & MTCR & MTACK
# BURST & FCS;

```


A.1.3 Memory Timing PAL

This device controls RAS and CAS timing, /DTACK generation, and high order RAM banking. This must be programmed into a 10ns 20L8 or equivalent device.

```
PARTNO      U300 ;
NAME        U300 ;
DATE        May 30, 1990 ;
REV         5 ;
DESIGNER    Dave Haynie ;
COMPANY     Commodore-Amiga ;
ASSEMBLY    BIGRAM ;
LOCATION     U300 ;

/*****
/*
/* Zorro III BIGRAM DRAM Timing
/*
/* This device controls the standard and refresh timing of the
/* dynamic RAM. Big-Time asynchronicity ahead! This also controls
/* banking within a CAS controlled memory bank.
/*
*****/
/*
/* DEVICE DATA:
/*
/* Device:          20L8-10
/* Clock:           NONE
/* Unused:          NONE
/*
*****/

/* INPUTS: */

PIN 1  =      !RASDEL      ;      /* RAS strobe delay */
PIN 2  =      !MUX         ;      /* DRAM Address multiplexer */
PIN 3  =      !MTCR        ;      /* Multiple cycle request */
PIN 4  =      !BURST       ;      /* We're in burst mode. */
PIN 5  =      !DOE         ;      /* Data time */
PIN 6  =      !SLAVE       ;      /* The board is responding */
PIN 7  =      !REFACK      ;      /* We're servicing a refresh request */
PIN 8  =      !SCRAM       ;      /* We're using static column RAM. */
PIN 9  =      A23          ;      /* System addresses */
PIN 10 =      A22          ;
PIN 11 =      A21          ;
PIN 13 =      A20          ;
PIN 14 =      MEG4         ;      /* 4 Meg parts? */
PIN 23 =      !CASDEL      ;      /* CAS strobe delay */

/* OUTPUTS: */

PIN 16 =      !CASN        ;      /* CAS strobe enable */
PIN 17 =      !CASOUT      ;      /* CAS delay input */
PIN 18 =      !REFCAS      ;      /* CAS for refresh */
PIN 19 =      !REFCYC      ;      /* We're in a refresh cycle. */
PIN 20 =      !DTACK       ;      /* Data is valid on bus */
PIN 21 =      !RASN        ;      /* RAS strobe enable */
PIN 22 =      BK0          ;      /* Small Bank bit 0 */
PIN 15 =      BK1          ;      /* Small Bank bit 1 */

/** OUTPUT TERMS: **/

/* The data valid signal. Data is valid on the bus if we're not in a refresh
the board is selected, and something's happened. The burst cycle is timed by
CAS delay only. */

DTACK      = SLAVE & !BURST & !REFACK & !REFCYC & DOE & RASDEL & CASDEL
            # SLAVE & !BURST & DTACK
            # SLAVE & BURST & !REFACK & !REFCYC & DOE & CASOUT & CASDEL & MTCR
            # SLAVE & BURST & DTACK & MTCR;

DTACK.OE   = SLAVE;

/* The RAS enable strobe. If we're not in refresh, it goes as soon
as we're sure the board is selected. If refresh is called for,
start a RAS cycle after the CAS delay. */

RASN       = !REFACK & !REFCYC & !RASDEL & !CASN & SLAVE
            # !REFACK & !REFCYC & RASN & !CASN & SLAVE
            # !REFACK & !REFCYC & RASN & CASN & !BURST & !SCRAM & !DTACK
            # !REFACK & !REFCYC & RASN & SLAVE & BURST
            # !REFACK & !REFCYC & RASN & SLAVE & SCRAM
            # REF CYC & CASDEL & !RASDEL
            # REF CYC & RASN & !RASDEL;
```

```

/* The CAS enable works differently for burst vs. non-burst. For non-burst,
it follows RASEN after DOE and MUX are asserted. In a burst cycle, it
follows MTCR. For refresh, CAS can't be enabled until we're sure that
RASDEL is negated, thus ensuring RAS precharge when a refresh cycle
immediately follows a standard memory cycle. */

```

```

CASOUT      = !REFACK & !REFCYC & !BURST & RASEN & MUX & DOE & !RASDEL
             # !REFACK & !REFCYC & !BURST & CASOUT & DTACK & SLAVE
             # !REFACK & !REFCYC & BURST & !CASDEL & MTCR
             # !REFACK & !REFCYC & BURST & CASOUT & MTCR
             # REFACK & REFCYC & !RASEN & !RASDEL
             # CASOUT & REFCYC & !RASEN;

```

```

/* The actual CAS that goes out is modified by our use of SCRAMs. If
SCRAMs are in use, CASEN goes low and stays low, while CASOUT works
the DTACK line. Otherwise, CASEN and CASOUT are the same. */

```

```

CASEN       = !REFACK & !REFCYC & !SCRAM & CASOUT
             # !REFACK & !REFCYC & SCRAM & CASOUT
             # !REFACK & !REFCYC & SCRAM & CASEN & SLAVE;

```

```

/* This is the rest of the refresh machine. A refresh cycle starts with a
valid refresh acknowledge and the assertion of the standard and refresh
CAS. RAS for refresh is asserted one CASDEL later, and standard CAS is
negated at the same point. The refresh counter will clear REFREQ when
REFCYC is asserted, and clear REFACK when REFREQ is negated. */

```

```

REFCAS      = REFACK & REFCYC & !CASDEL & !RASDEL
             # REFCAS & REFCYC & !MUX
             # REFCAS & REFCYC & RASEN & !RASDEL;

```

```

REFCYC      = REFACK & !CASDEL & !RASDEL
             # REFCYC & CASOUT & !RASDEL
             # REFCYC & RASEN
             # REFCYC & RASDEL;

```

```

/* Bank control. The bank is controlled by A23 and A22 for 4 Meg memory,
A21 and A20 for 1 Meg memory. */

```

```

BK0         = A22 & MEG4
             # A20 & !MEG4;

```

```

BK1         = A23 & MEG4
             # A21 & !MEG4;

```

A.1.4 CAS Control PAL

This device controls the CAS generation and banking. This must be programmed into a 15ns 20L8 PAL device or equivalent.

```

PARTNO      U304 ;
NAME        U304 ;
DATE        May 30, 1990 ;
REV         3 ;
DESIGNER    Dave Haynie ;
COMPANY     Commodore-Amiga ;
ASSEMBLY    BIGRAM ;
LOCATION     U304 ;

/*****
/*
/* Zorro III BIGRAM DRAM CAS Select
/*
/* This device controls the CAS strobes, which control DRAM byte
/* enables and most significant bank.
/*
/*
*****/
/*
/* DEVICE DATA:
/*
/* Device:          20L8-15
/* Clock:           NONE
/* Unused:          NONE
*****/

/* INPUTS: */

PIN 1  =      !CASEN      ;      /* Normal CAS enable */
PIN 2  =      !DTACK      ;      /* Zorro III cycle termination */
PIN 3  =      !REFCAS     ;      /* CAS for refresh cycle */
PIN 4  =      !REFACK     ;      /* We're in refresh */
PIN 5  =      !DS3        ;      /* Zorro III data strobes */
PIN 6  =      !DS2        ;
PIN 7  =      !DS1        ;
PIN 8  =      !DS0        ;
PIN 9  =      SCRAM        ;      /* Using static column memories */
PIN 10 =      READ        ;      /* Zorro III Read enable */
PIN 11 =      MEG4        ;      /* Are we using 4 Meg parts? */
PIN 13 =      !CADDR      ;      /* Column Address Valid */
PIN 14 =      A24         ;      /* Address lines */
PIN 23 =      A22         ;

/* OUTPUTS: */

PIN 15 =      !CASL0      ;      /* Lower bank CAS */
PIN 16 =      !CASL1      ;
PIN 17 =      !CASL2      ;
PIN 18 =      !CASL3      ;
PIN 19 =      !CASH0      ;      /* Upper bank CAS */
PIN 20 =      !CASH1      ;
PIN 21 =      !CASH2      ;
PIN 22 =      !CASH3      ;

/** INTERNAL TERMS: **/

/* The CAS lines are the highest order banking control. If we're using 1 Meg
parts, lower is $0000000-$03ffff, upper is $0400000-$07ffff, so A22 controls
the banking. If we're using 4 Meg parts, lower is $0000000-$0fffff, upper is
$1000000-$1fffff, so A24 controls the banking. */

lower      = !A24 & MEG4 & CASEN & CADDR
# !A22 & !MEG4 & CASEN & CADDR;
upper      = A24 & MEG4 & CASEN & CADDR
# A22 & !MEG4 & CASEN & CADDR;

/** OUTPUT TERMS: **/

/* The CAS terms are simple. There are two banks of memory, and the banking
is controlled as above. On writes, the data strobes control the particular
CAS line, and we wait for WRDEL so that data is guaranteed valid on the
DRAM bus. On reads, all CAS lines in a bank are asserted ASAP. On
refresh, all CAS lines are asserted. */

CASL0      = lower & !READ & DS0
# lower & READ
# REFCAS;

CASL1      = lower & !READ & DS1
# lower & READ
# REFCAS;

```

```

CASL2      = lower & !READ & DS2
            # lower & READ
            # REFCAS;

CASL3      = lower & !READ & DS3
            # lower & READ
            # REFCAS;

CASH0      = upper & !READ & DS0
            # upper & READ
            # REFCAS;

CASH1      = upper & !READ & DS1
            # upper & READ
            # REFCAS;

CASH2      = upper & !READ & DS2
            # upper & READ
            # REFCAS;

CASH3      = upper & !READ & DS3
            # upper & READ
            # REFCAS;

```

A.1.5 Refresh Counter PAL

This device is responsible for timing the CAS-before-RAS refresh used by the DRAM system. This must be programmed into a 25ns 16R8 or equivalent device.

```

PARTNO      U306 ;
NAME        U306 ;
DATE        May 30, 1990 ;
REV         1 ;
DESIGNER    Dave Haynie ;
COMPANY     Commodore-Amiga ;
ASSEMBLY    BIGRAM ;
LOCATION     U306 ;

/*****
/*
/* Zorro III BIGRAM DRAM Refresh Counter.
/*
/* This device is responsible for generating refresh request.
/*
*****/
/*
/* DEVICE DATA:
/*
/* Device: 16R8-25
/* Clock: C7M
/* Unused: NONE
*****/

/* INPUTS: */

PIN 2 = !REFACK ; /* We're servicing a refresh request */
PIN 3 = !REFCYC ; /* We're in a refresh cycle. */

/* BIDIRECTIONALS: */

PIN 19 = !REFREQ ; /* Refresh request */

/* USED INTERNALLY: */

PIN 18 = !R0 ; /* Counter bits */
PIN 17 = !R1 ;
PIN 16 = !R2 ;
PIN 15 = !R3 ;
PIN 14 = !R4 ;
PIN 13 = !R5 ;
PIN 12 = !R6 ;

/** INTERNAL TERMS: **/

field count = [R6..0];

/** OUTPUT TERMS: **/

/* The refresh request is asserted when the terminal count has been reached.
It's held until REFHOLD is asserted. */

REFREQ.D = count:70
          # REFREQ & !REFCYC;

/* The refresh counter is pretty simple. We're assuming one refresh cycle
every 15,625ns, which works out fine for the 8ms, 512 row 1 Meg parts
or the 16ms, 1024 row 4 Meg parts. However, the maximum TRAS period
is only 10,000ns, which must be taken into account to support burst
mode. Counting 71 140ns C7M clocks gets me to 9,940ns, close enough.
The counter resets when REFCYC comes along. */

R0.D = !REFCYC & !R0;
R1.D = !REFCYC & R0 & !R1
      # !REFCYC & !R0 & R1;
R2.D = !REFCYC & R0 & R1 & !R2
      # !REFCYC & !R1 & R2
      # !REFCYC & !R0 & R2;
R3.D = !REFCYC & R0 & R1 & R2 & !R3
      # !REFCYC & !R2 & R3
      # !REFCYC & !R1 & R3
      # !REFCYC & !R0 & R3;
R4.D = !REFCYC & R0 & R1 & R2 & R3 & !R4
      # !REFCYC & !R3 & R4
      # !REFCYC & !R2 & R4

```

```

# !REFCYC & !R1 & R4
# !REFCYC & !R0 & R4;

R5.D
= !REFCYC & R0 & R1 & R2 & R3 & R4 & !R5
# !REFCYC & !R4 & R5
# !REFCYC & !R3 & R5
# !REFCYC & !R2 & R5
# !REFCYC & !R1 & R5
# !REFCYC & !R0 & R5;

R6.D
= !REFCYC & R0 & R1 & R2 & R3 & R4 & R5 & !R6
# !REFCYC & !R5 & R6
# !REFCYC & !R4 & R6
# !REFCYC & !R3 & R6
# !REFCYC & !R2 & R6
# !REFCYC & !R1 & R6
# !REFCYC & !R0 & R6;

```

A.2 Schematics

The following pages contain the schematics for the example memory board. The list of parts is as follows:

Capacitors

0.01 μ F MLC	C109
0.10 μ F MLC	C100-C107,C200-C203,C300-C306,C400-C404
0.33 μ F MLC	C500,C502,C504,C506,C508,C510,C512,C514,C516,C518,C520, C522,C524,C526,C528,C530,C600,C602,C604,C606,C608,C610, C612,C614,C616,C618,C620,C622,C624,C626,C628,C630
47 μ F, 16V Electro	C108
100 μ F, 16V Electro	C110,C111

Resistors

22 Ω , 5%, 1/4 Watt	R300,R301
1K Ω , 5%, 1/4 Watt	R100

Resistor Packs

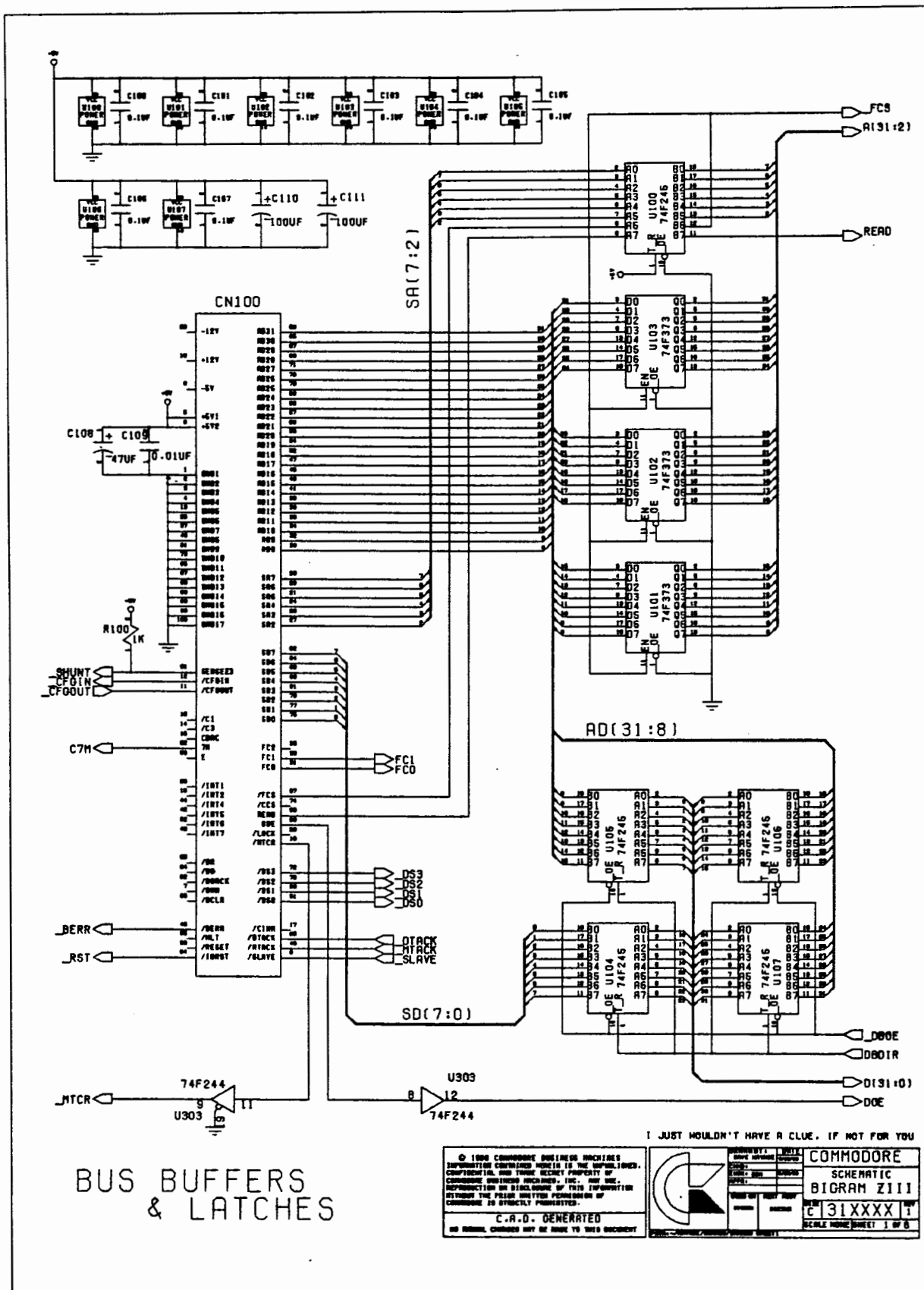
22 Ω , 4x8	RP300-RP303,RP400-RP404
1K Ω , 9x10	RP100

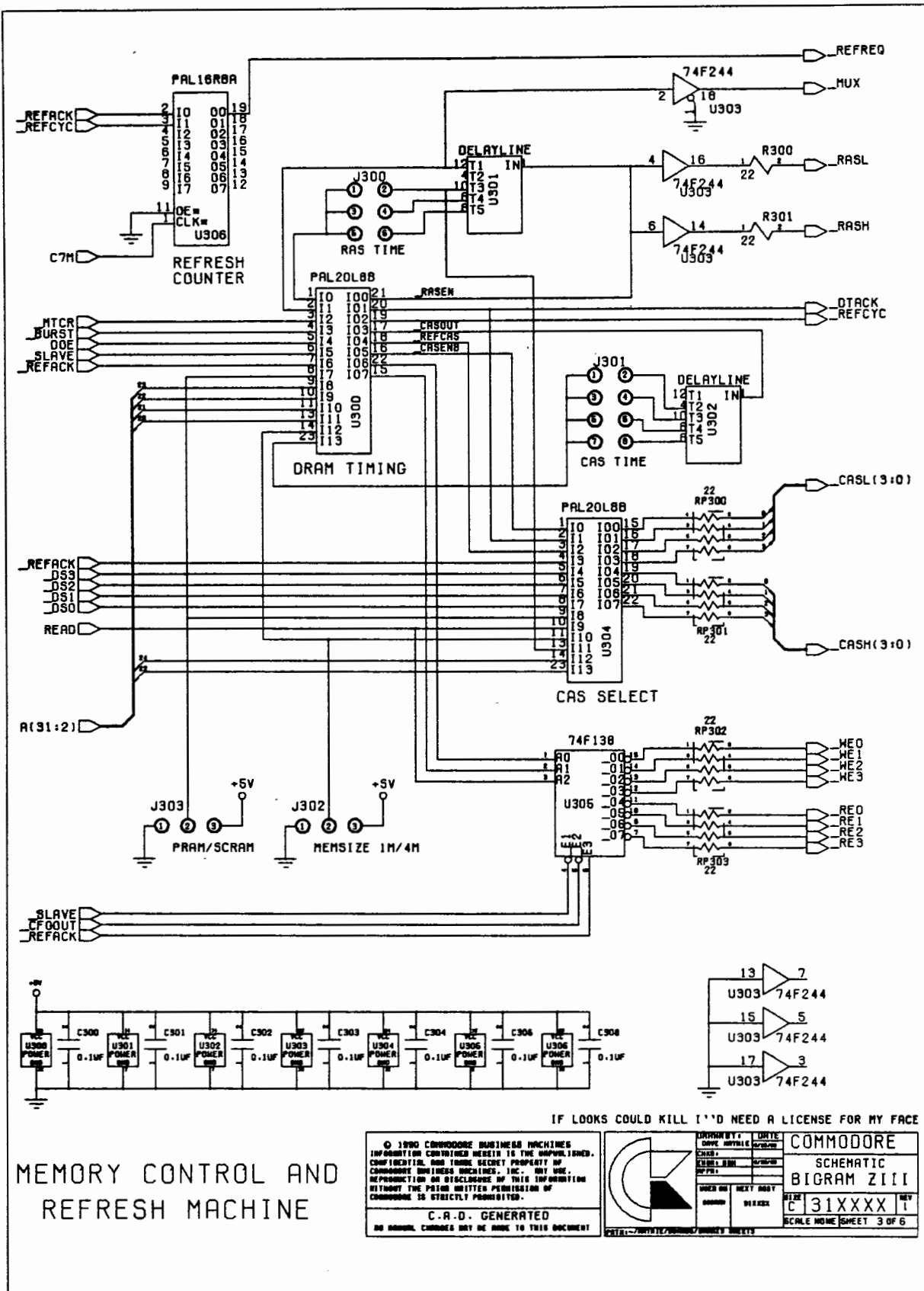
Post Jumpers

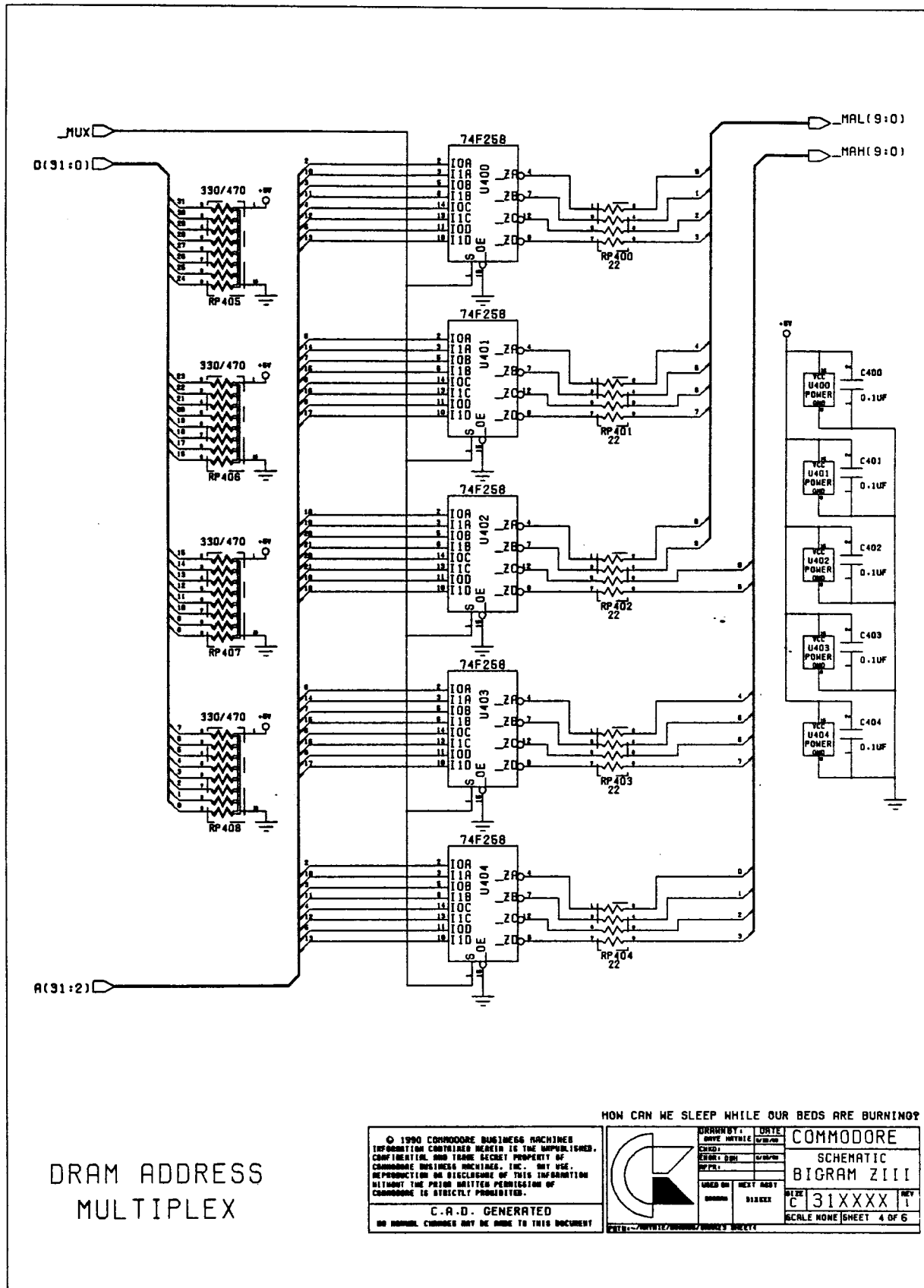
3 Pin, 0.100	J200,J302,J303
2 Pin x 3 Pin, 0.100	J300
2 Pin x 4 Pin, 0.100	J301

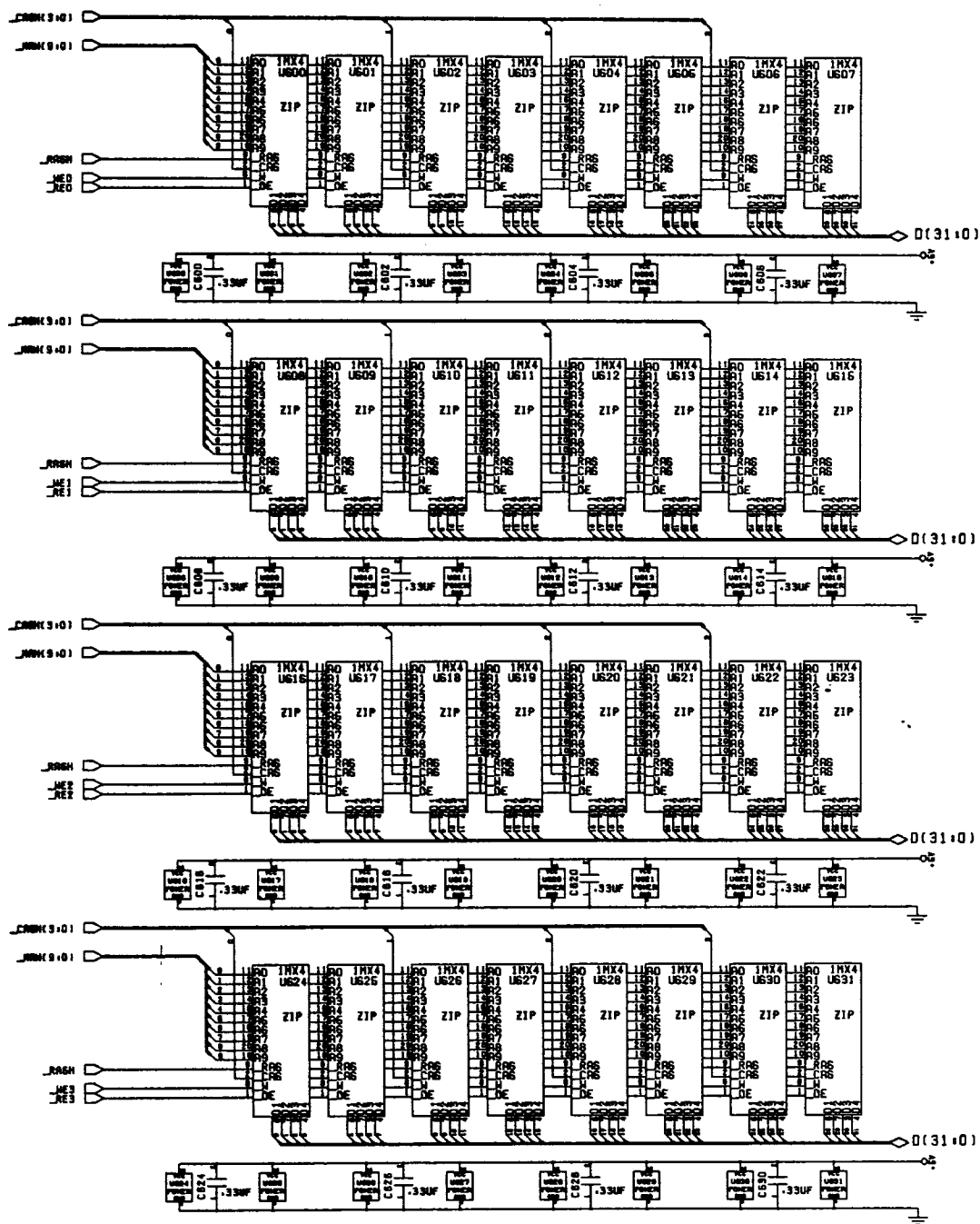
Integrated Circuits

74F138	U305
74F244	U303
74F245	U100,U104-U107
74F258	U400-U404
74F373	U101-U103
74F374	U202
74F521	U203
PAL 16L8B	U200
PAL 16R8A	U306
PAL 20L8B	U300,U304
PAL 20L8D	U201
Tap Delay 100ns	U301
Tap Delay 50ns	U302
DRAM 256K x 4, 80ns or 1M x 4, 80ns	U500-U531,U600-U631









HIGH MEMORY BANK

I CAN'T REMEMBER. I CAN'T RECALL. I HAVE MEMORY OF ANYTHING AT ALL.

<p>© 1989 COMMODORE INTERNATIONAL INC. ALL RIGHTS RESERVED. INFORMATION CONTAINED HEREIN IS THE PROPERTY OF COMMODORE INTERNATIONAL INC. AND IS NOT TO BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC OR MECHANICAL, INCLUDING PHOTOCOPYING, RECORDING, OR BY ANY INFORMATION STORAGE AND RETRIEVAL SYSTEM, WITHOUT PERMISSION IN WRITING FROM COMMODORE INTERNATIONAL INC.</p> <p>C.A.D. GENERATED</p> <p>NO OTHER COMPANY MAY BE MADE IN YOUR COUNTRY</p>	<p>COMMODORE</p> <p>BIORAH ZIII</p> <p>E31XXXX</p> <p>SCALE 1:1000000 8 OF 8</p>
---	--

A.3 Zorro III Configuration

While presumably AmigaOS 2.0 will understand Zorro III AUTOCONFIG conventions, the following routine is useful for configuring simple Zorro III boards in an AmigaOS 1.3 system. Note that many popular MMU configurations don't map in the Zorro III configuration space at \$FF000000, so this program is not likely to work with an MMU mapping in place.

```
/* ===== */
/* A very simple configuration utility for Zorro III boards. This code will
   configure Zorro III cards that are placed after any Zorro II cards in
   the A3000. All configuration is done based on 16 meg slots and no magic
   for autoboot, etc. Eventually 2.0 will do this better. */

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/configregs.h>
#include <libraries/configvars.h>
#include <libraries/expansionbase.h>
#include <stdio.h>
#include <ctype.h>
#include <functions.h>

/* ===== */

/* Modified configuration information. */

/* Extensions to the TYPE field. */

#define E_Z3EXPBASE      0xff000000L
#define E_Z3EXPSTART     0x10000000L
#define E_Z3EXPFINISH    0x7fffffffL
#define E_Z3SLOTSIZE     0x01000000L
#define E_Z3ASIZEINC     0x00010000L

#define ERT_ZORROII      ERT_NEWBOARD
#define ERT_ZORROIII     0x8U

/* Extensions to the FLAGS field. */

#define ERFB_EXTENDED     5L
#define ERFF_EXTENDED     (1L<<5)

static BoardSize[2][8] = {
    { 0x00800000, 0x00010000, 0x00020000, 0x00040000,
      0x00080000, 0x00100000, 0x00200000, 0x00400000 },
    { 0x01000000, 0x02000000, 0x04000000, 0x08000000,
      0x10000000, 0x20000000, 0x40000000, 0x00000000 }
};

#define ERFB_QUICKVALID    4L
#define ERFF_QUICKVALID    (1L<<4)

#define ERF_SUBMASK       0x0fL
#define ERF_SUBSAME       0x00L
#define ERF_SUBAUTO       0x01L
#define ERF_SUBFIXED      0x02L
#define ERF_SUBRESERVE    0x0eL

static SubSize[16] = {
    0x00000000, 0x00000000, 0x00010000, 0x00020000,
    0x00040000, 0x00080000, 0x00100000, 0x00200000,
    0x00400000, 0x00600000, 0x00800000, 0x00a00000,
    0x00c00000, 0x00e00000, 0x00000000, 0x00000000
};

#define PRVB(x)if (verbose) { printf(x); }

static BOOL      verbose = TRUE;
static BOOL      anyone = FALSE;
struct ExpansionBase *ExpansionBase;
static ULONG     Z3Space = 0x10000000L;

/* ===== */

/* These functions are involved in finding a Zorro III board. */

/* This function reads the logical value stored at the given Zorro III
   ROM location. This corrects for complements and the differing offsets
   depending on location. */
```

```

UBYTE ReadZ3Reg(base, reg)
WORD *base;
WORD reg;
{
    ULONG *Z3base;
    UWORD result;

    if (base == (WORD *)E_EXPANSIONBASE) {
        base += (reg>>1);
        result = ((*base++)&0xf000)>>8;
        result = ((*base)&0xf000)>>12;
    } else {
        Z3base = (ULONG *) (base+(reg>>1));
        result = ((*Z3base)&0xf0000000)>>24;
        result |= ((*Z3base+0x40)&0xf0000000)>>28;
    }
    if (reg) result = ~result;

    return (UBYTE)result;
}

/* This function types the board in the system, returning the type code.
   There are four possibilities -- no board, a Zorro II board, a Zorro III
   board at the Zorro II configuration slot, and a Zorro III board at the
   Zorro III configuration slot. */

#define BT_NONE      0
#define BT_Z2       1
#define BT_Z3_AT_Z2 2
#define BT_Z3_AT_Z3 3

BYTE TypeOfPIC() {
    UBYTE type;
    UWORD manf;

    type = ReadZ3Reg(E_EXPANSIONBASE, 0x00);
    manf = ReadZ3Reg(E_EXPANSIONBASE, 0x10)<<8 | ReadZ3Reg(E_EXPANSIONBASE, 0x14);

    if (manf != 0x0000 && manf != 0xffff) {
        if ((type & ERT_TYPEMASK) == ERT_ZORROII) return BT_Z2;
        if ((type & ERT_TYPEMASK) == ERT_ZORROIII) return BT_Z3_AT_Z2;
    }
    type = ReadZ3Reg(E_Z3EXPBASE, 0x00);
    manf = ReadZ3Reg(E_Z3EXPBASE, 0x10)<<8 | ReadZ3Reg(E_Z3EXPBASE, 0x14);

    if (manf != 0x0000 && manf != 0xffff)
        if ((type & ERT_TYPEMASK) == ERT_ZORROIII) return BT_Z3_AT_Z3;

    return BT_NONE;
}

/* This function fills the configuration ROM field of the given
   ConfigDev, form the given address, based on the appropriate mapping
   rules. */

void InitZ3ROM(base, cd)
WORD *base;
struct ConfigDev *cd;
{
    struct ExpansionRom *rom;

    rom = &cd->cd_Rom;

    rom->er_Type = ReadZ3Reg(base, 0x00);
    rom->er_Product = ReadZ3Reg(base, 0x04);
    rom->er_Flags = ReadZ3Reg(base, 0x08);
    rom->er_Reserved03 = ReadZ3Reg(base, 0x0c);
    rom->er_Manufacturer = ReadZ3Reg(base, 0x10)<< 8 | ReadZ3Reg(base, 0x14);
    rom->er_SerialNumber = ReadZ3Reg(base, 0x18)<<24 | ReadZ3Reg(base, 0x1c)<<16 |
        ReadZ3Reg(base, 0x20)<< 8 | ReadZ3Reg(base, 0x24);
    rom->er_InitDiagVec = ReadZ3Reg(base, 0x28)<< 8 | ReadZ3Reg(base, 0x2c);
    rom->er_Reserved0c = ReadZ3Reg(base, 0x30);
    rom->er_Reserved0d = ReadZ3Reg(base, 0x34);
    rom->er_Reserved0e = ReadZ3Reg(base, 0x38);
    rom->er_Reserved0f = ReadZ3Reg(base, 0x3c);
}

/* This function locates a Zorro III board. If it finds one in the
   unconfigured state, it allocates a ConfigDev for it, fills in the
   configuration data, and returns that ConfigDev. Otherwise it returns
   NULL. It knows the basics of what to do should it encounter a
   Zorro II board sitting in the way. */

struct ConfigDev *FindZ3Board() {
    struct ConfigDev *cd;

    while (TRUE) {
        if (!(cd = AllocConfigDev())) return NULL;
    }
}

```

```

switch (TypeOfPIC()) {
    case BT_NONE :
        FreeConfigDev(cd);
        return NULL;
    case BT_Z2 :
        PRVB("FOUND: Z2 Board, Configuring");
        if (!ReadExpansionRom(E_EXPANSIONBASE,cd))
            if (!ConfigBoard(E_EXPANSIONBASE,cd))
                AddConfigDev(cd);
        anyone = TRUE;
        break;
    case BT_Z3_AT_Z2 :
        PRVB("FOUND: Z3 Board (Z2 Space), Configuring");
        InitZ3ROM(E_EXPANSIONBASE,cd);
        cd->cd_BoardAddr = (APTR)E_EXPANSIONBASE;
        anyone = TRUE;
        return cd;
    case BT_Z3_AT_Z3 :
        PRVB("FOUND: Z3 Board (Z3 Space), Configuring");
        InitZ3ROM(E_Z3EXPBASE,cd);
        cd->cd_BoardAddr = (APTR)E_Z3EXPBASE;
        anyone = TRUE;
        return cd;
}
return NULL;
}

/* ===== */
/* These functions are involved in configuring a Zorro III board. */
/* This function writes the configuration address stored in the given
   ConfigDev to the board in the proper way. */

void WriteCfgAddr(base,cd)
UWORD *base;
struct ConfigDev *cd;
{
    UBYTE nybreg[4],bytereg[2],*bytebase;
    UWORD wordreg,i,*wordbase;

    wordreg = (((ULONG)cd->cd_BoardAddr)>>16);
    bytereg[0] = (UBYTE)(wordreg & 0x00ff);
    bytereg[1] = (UBYTE)(wordreg >> 8);
    nybreg[0] = ((bytereg[0] & 0x0f)<<4);
    nybreg[1] = ((bytereg[0] & 0xf0));
    nybreg[2] = ((bytereg[1] & 0x0f)<<4);
    nybreg[3] = ((bytereg[1] & 0xf0));

    bytebase = (UBYTE *) (base + 22);
    wordbase = (UWORD *) (base + 22);

    if (base == (UWORD *)E_EXPANSIONBASE) {
        (*(bytebase+0x002)) = nybreg[2];
        (*(bytebase+0x000)) = bytereg[1];
        (*(bytebase+0x006)) = nybreg[1];
        (*(bytebase+0x004)) = bytereg[0];
    } else {
        (*(bytebase+0x104)) = nybreg[0];
        (*(bytebase+0x004)) = bytereg[0];
        (*(bytebase+0x100)) = nybreg[2];
        (*(wordbase+0x000)) = wordreg;
    }
}

/* This function automatically sizes the configured board described by the
   given ConfigDev. It doesn't attempt to preserve the contents. */

void AutoSizeBoard(cd)
struct ConfigDev *cd;
{
    ULONG i,realmax,logicalsize = 0;

    realmax = ((ULONG)cd->cd_SlotSize) * E_Z3SLOTSIZE + (ULONG)cd->cd_BoardAddr;

    for (i = (ULONG)cd->cd_BoardAddr; i < realmax; i += E_Z3ASIZEINC)
        *((ULONG *)i) = 0;

    for (i = (ULONG)cd->cd_BoardAddr; i < realmax; i += E_Z3ASIZEINC) {
        if (*(ULONG *)i) != 0 break;
        *((ULONG *)i) = 0xaa5500ff;
        if (*(ULONG *)i) != 0xaa5500ff break;
        logicalsize += E_Z3ASIZEINC;
    }
    cd->cd_BoardSize = (APTR)logicalsize;
}

```



```

/* This function configures a Zorro III board, based on the initialization
   data in its ConfigDev structure. */

void ConfigZ3Board(cd)
struct ConfigDev *cd;
{
    APTR base = cd->cd BoardAddr;
    UWORD sizecode, extended, subsize;
    ULONG physsize, logsize;
    char *memname;

    /* First examine the physical sizing of the board. */

    sizecode = cd->cd Rom.er_Type & ERT MEMSIZE;
    extended = ((cd->cd Rom.er_Flags & ERFF_EXTENDED) != 0);

    physsize = BoardSize[extended][sizecode];

    cd->cd BoardAddr = (APTR)Z3Space;
    cd->cd BoardSize = (APTR)physsize;
    cd->cd SlotAddr = (Z3Space-E_Z3EXPSTART)/E_Z3SLOTSIZE;
    cd->cd SlotSize = ((physsize/E_Z3SLOTSIZE)>0)?(physsize/E_Z3SLOTSIZE):1;
    Z3Space += cd->cd SlotSize * E_Z3SLOTSIZE;

    /* Next, process the sub-size, if any. */

    if (subsize = (cd->cd Flags & ERF SUBMASK))
        cd->cd BoardSize = (APTR)SubSize[subsize];

    if (verbose) {
        printf("  BOARD STATS:");
        printf("      ADDRESS: %lx", cd->cd BoardAddr);
        if (cd->cd BoardSize)
            printf("      SIZE: %lx", cd->cd BoardSize);
        else
            printf("      SIZE: AUTOMATIC => ");
    }

    /* Now, configure the board. */

    WriteCfgAddr(base, cd);
    if (!cd->cd BoardSize) {
        AutoSizeBoard(cd);
        printf("%lx", cd->cd BoardSize);
    }

    if (cd->cd BoardSize && (cd->cd Rom.er_Type & ERTF MEMLIST)) {
        strcpy(memname = (char *)AllocMem(20L, MEMF_CLEAR), "Zorro III Memory");
        AddMemList(cd->cd BoardSize, MEMF_FAST|MEMF_PUBLIC, 10, cd->cd BoardAddr, memname);
    }

    AddConfigDev(cd);
}

/* ===== */

/* This is the main program. */

void main(argc, argv)
int argc;
char *argv[];
{
    int i;
    struct ConfigDev *cd;

    if (!(ExpansionBase = (struct ExpansionBase *)OpenLibrary("expansion.library", 0L))) {
        printf("Error: Can't open \"expansion.library\"");
        exit(10);
    }

    if (argc > 1)
        for (i = 1; i < argc; ++i) switch (toupper(argv[i][0])) {
            case 'Q': verbose = FALSE; break;
            case 'V': verbose = TRUE; break;
        }

    while (cd = FindZ3Board()) ConfigZ3Board(cd);

    if (!anyone) PRVB("No PICs left to configure");
    CloseLibrary((struct ExpansionBase *)ExpansionBase);
}

```




ARCNET

by Joseph E. Augenbraun

Introduction

ARCNET is designed to be a low cost computer network. It works on a modified token passing scheme, has a data rate of 2.5 M bits per second and automatically reconfigures itself as nodes are added or removed from the network. Every node has an ID number associated with it, the node number being set on a row of 8 DIP switches. ARCNET supports up to 255 nodes.

Below is a table comparing ARCNET to several other popular networking alternatives. It shows that while Ethernet is significantly faster, ARCNET is less expensive. It also shows that while the cost of LocalTalk/AppleTalk is low, the speed is extremely low too. This makes ARCNET competitive for applications where the cost of Ethernet is prohibitive, but good performance is necessary.

Network Hardware	LocalTalk	ARCNET	Ethernet
Popular network software	AppleTalk	Novell	TCP/IP
Speed (Mbit/second)	0.3	2.5	10
Cost	Low	Medium	High
Typical Installation	Macintosh	IBM PC	Minicomputers

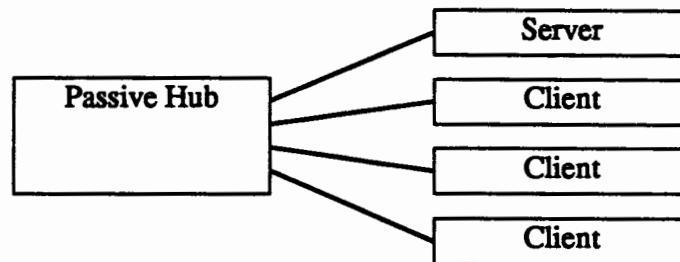
There are many different kinds of software that may be run on any given type of network hardware; the table gives the most popular. Likewise, the typical installation item is only there to give an idea of where each network became most popular. There is now a trend towards total separation of network hardware and software (a sort of a mix and match approach). Thus you have Apple Macintosh installations running AppleTalk over Ethernet, IBM/PC's running Novell over ARCNET, or Commodore's planned implementation of TCP/IP for ARCNET.

Interfaces

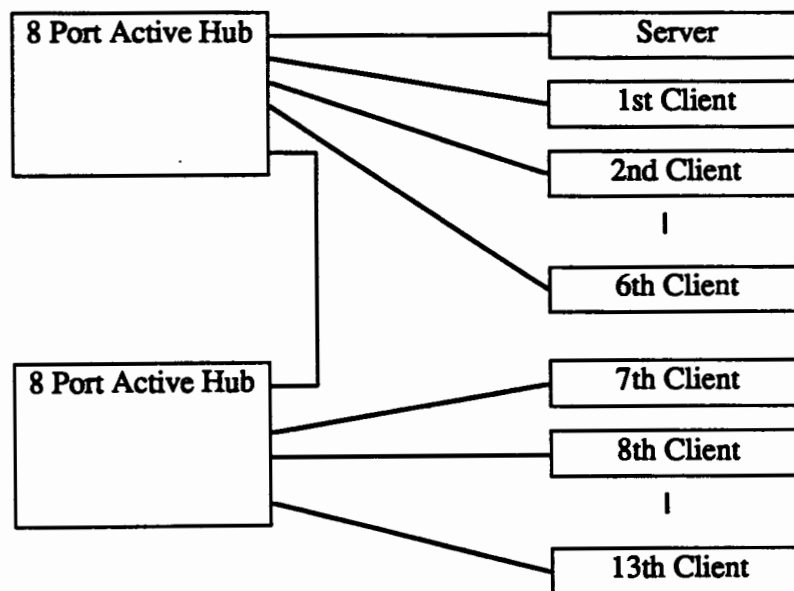
ARCNET allows for different physical transmission mediums, that is "interfaces". No matter which interface is used, protocols remain the same, the interfaces simply differ in electrical and topological characteristics. The following interfaces are available for ARCNET:

Star type (sometimes referred to as LAND)

The star type network uses 93 ohm coax to connect nodes together. Each node on a star network is connected to a dedicated port on a device called a "hub". If there is need for more devices than ports available on the hub, another hub must be purchased. Communication is typically via a 15 volt peak to peak voltage, and there may be up to 2000 feet between any 2 nodes. This was the original interface available for ARCNET..

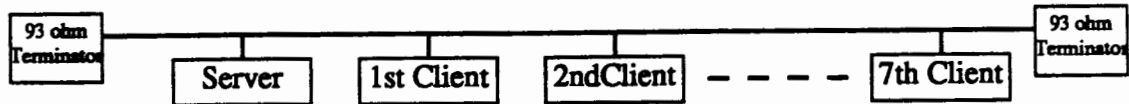


When setting up a star network, devices are connected together using either a "passive hub" or an "active hub." Passive hubs are inexpensive, but are only used for up to 4 nodes. If more nodes are needed in a star network, active hubs must be used. They are available with different numbers of ports, and can be ganged together, or can be connected as a unit on another active hub, as shown below. Note that use of multiple active hubs reduces the 2000 foot maximum distance between most distant nodes.



Bus type (sometimes referred to as HIT)

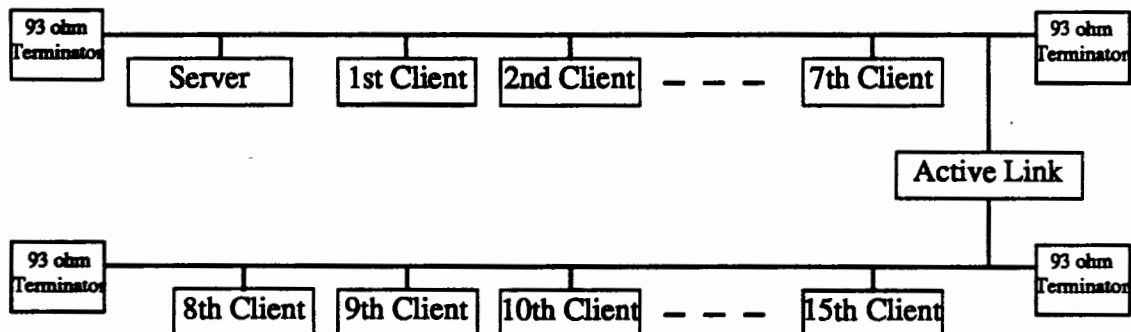
The bus type network also uses 93 ohm coax to connect nodes together, but the ends of the cable must also be terminated with 93 ohm terminators. Up to 8 nodes may be connected



together via BNC T connectors as long as the cable length is no greater than 1000 feet. Other lengths and numbers of nodes may be used as shown in the table below.

Number of nodes	Max cable length	Number of nodes	Max cable length
2	1428 feet	11	783 feet
3	1356 feet	12	711 feet
4	1285 feet	13	640 feet
5	1213 feet	14	568 feet
6	1141 feet	15	496 feet
7	1070 feet	16	425 feet
8	998 feet	17	353 feet
9	926 feet	18	281 feet
10	855 feet	19	210 feet

The number of nodes on a network may be increased by adding things known as "Active Links". These are devices that have 2 BNC connectors on them, and can be thought of as repeaters from one bus network to another. The active link counts as a node on each of the 2 buses that it is connecting together.



Twisted pair

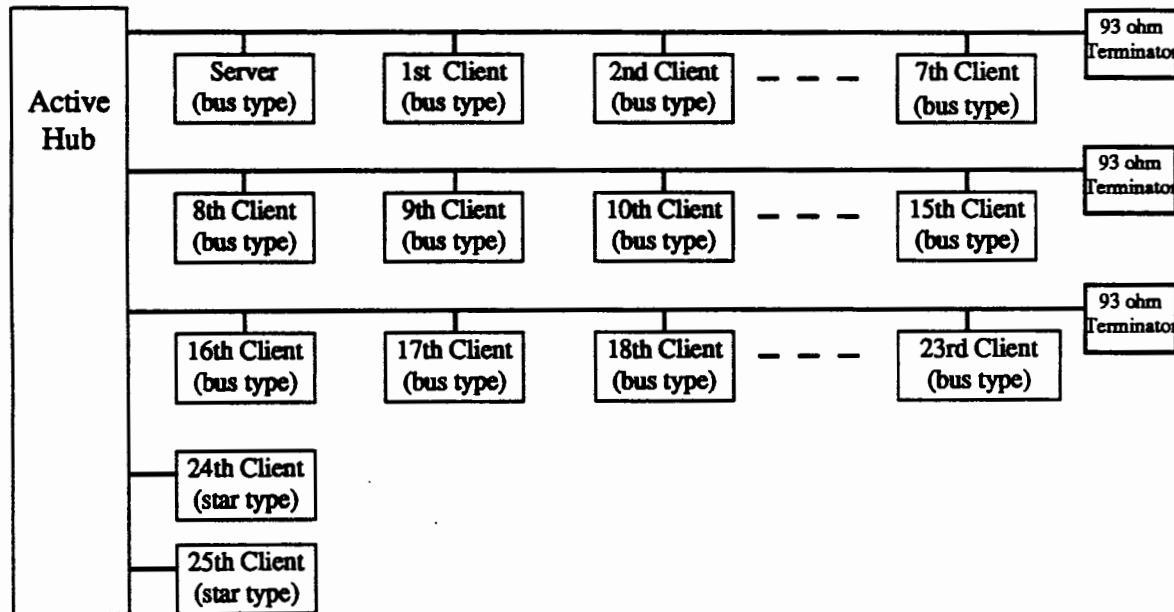
The twisted pair interface is designed to use ordinary twisted pair cable that is often already available in many large buildings. Network cost is reduced because special purpose coax cable does not have to be run.

Fiber optic

The fiber optic interface uses light as the network medium. The topology is similar to the star type network. The advantages of the fiber optic interface is noise immunity and longer distances between nodes being possible.

Combination of Bus and Star

Bus and star type networks can be combined as long as some simple rules are followed. When a bus type device and a star type device are on the same cable, the star device must never be turned off, the star device should be at the end of the cable (no terminator is used), and the use of passive hubs is not allowed. The distance from the bus device to the furthest node can be no greater than 1500 feet. These rules allow bus type devices to be connected to active hubs and star type devices, as shown in the example below:



Network Protocol

Data transfer in ARCNET is based on a modified token passing scheme. A token passing scheme is one in which a magic cookie, called a token, is passed from one node to another until each node has had the token, and the token goes back to the first node. The only node which may send a message is the one that currently holds the token. ARCNET calls their token passing scheme "modified" because all passes of the token are acknowledged by the receiving node.

If a node wishes to send a packet of data it waits until it receives the token. Then the node transmits a free buffer enquiry message to the node it wishes to send data to. If the receiving node cannot accept the data packet, it transmits back a negative acknowledgement message, and the transmitting node gives up (it will try again when it next gets the token) and passes the token on to the next node in the chain. If the receiving node can accept the data packet, it transmits back an acknowledgement message, the transmitting node sends the data packet, and the receiving node verifies the data. If the data was successfully received, the receiving node transmits an acknowledge message. If it wasn't, the receiving node doesn't transmit anything. If the transmitting node doesn't receive an acknowledge it tries to resend the data packet. After this is done, the transmitting node passes the token to the next node on the chain.

Line Protocol

The lowest level of any digital communications medium is the transmission of "1"s and "0"s. ARCNET uses something called an "Isochronous" line protocol. This means that there is a constant time separating each data bit, whether it is a "1" or a "0". Each bit is transmitted in exactly one 400 ns clock interval. A "1" is indicated with a 200 ns pulse within the clock interval. A "0" is indicated by the lack of a pulse. Information is transmitted as 8 bit bytes. Each byte is preceded by 2 bits of logic "1" and followed by one bit of logic "0". Thus it takes eleven clock periods of 400 ns each or 4.4 μ s to transmit one byte. The line idles in the logic "0" condition. All transmissions start with something called an "alert burst". An alert burst is six contiguous intervals of logic "1". This allows the receiving node to phase lock its receiving clock before any real information is sent. There are five different kinds of information that may be sent by a node:

Invitation to transmit

The invitation to transmit passes the token from one node to another. It consists of an alert burst followed by the EOT (End Of Transmission, ASCII 0x4) character, followed by 2 repeated DID (Destination ID) characters. In this case the DID character is the ID number of the next node in the token chain.

Free buffer enquiries

The free buffer enquiries message is used to ask another node if it can accept data. It consists of an alert burst followed by the ENQ (ENquiry, ASCII 0x85) character, followed by 2 repeated DID characters. In this case the DID character is the ID number of the node that we want to send data to.

Data packet

This is the message that actually sends data. It consists of an alert burst followed by the SOH (Start Of Header, ASCII 0x1) character, followed by 2 repeated DID characters, which is the ID number of the node that we are trying to send data to. This is followed by either the COUNT character in the case of a short packet or ASCII 0x0 followed by the COUNT character in the case of a long packet. The COUNT character is related to the number of data bytes as follows:

Short Packet: COUNT = 256 - Number of data bytes

Long Packet: COUNT = 512 - Number of data bytes

This is followed by the data, followed by 2 CRC characters. The CRC polynomial is $X^{16} + X^{15} + X^2 + 1$.

Acknowledgement

This message is used to acknowledge the successful reception of a packet, or as an affirmative response to a free buffer enquiries message. The acknowledge message consists of the alert burst followed by ACK (ACKnowledge, ASCII 0x86).

Negative acknowledgement

This message is used as a negative response to a free buffer enquiries message. It consists of an alert burst followed by a NAK (Negative ACKnowledge, ASCII 0x15).

Network Reconfiguration

Whenever a new node is activated or an existing node doesn't receive the token for 840 ms, a NETWORK RECONFIGURATION is performed. The node initiating the NETWORK RECONFIGURATION starts by sending a RECONFIGURE BURST. This is eight logic "1"'s followed by one logic "0" repeated 765 times. This burst is longer than any other type of transmission, which causes the destruction of the token, and the termination of all other activity on the network.

The second phase of the NETWORK RECONFIGURATION starts when any node sees an idle line for more than 78.2 μ s, which only occurs when the token is lost, whether due to the first phase of NETWORK RECONFIGURATION or for some other reason.

In this phase each node starts an internal timeout of $146 \times (255 - \text{ID number}) \mu\text{s}$. The NID (Next ID) register on each node is set to its own ID number. If any given node doesn't see line activity before its timer times out, it tries to pass a token to a node with an ID equal to the currently stored NID. Within a given network, only the node with the highest ID number will try to do this. After generating the token, the node waits for line activity. If there is no activity for $74.7 \mu\text{s}$, the node increments the NID register (mod 256), and transmits a token with that new value. It continues doing this until it finds a node to pass the token to.

Once the token is successfully passed to the next node, this node releases the line, and the next node goes through the same exercise to figure out what the next node in the chain for it is. Each node then stores the number of the next node in the chain, so it can instantly pass the token to the next node in the chain without having to muck about with transmitting to nodes that don't exist.

The time required for NETWORK RECONFIGURATION depends on the number of nodes and cable lengths, but is normally in the range of 24 - 61 ms.

There is another facility available known as BROADCAST MESSAGES. A BROADCAST MESSAGE allows any given node to transmit data to all nodes simultaneously. ID number zero is reserved for this feature, so no node on the network can be assigned ID number zero.

Commodore Implementation

Commodore now has ARCNET hardware available for both the A500 and A2000/A3000. The A560 plugs onto the side of an A500, and comes with a replacement power supply for your A500 that provides the extra power required to drive the A560. The A560 has provisions to accept an optional internal 1 Mbyte memory expansion board. The A2060 plugs into any 100 pin Zorro II slot; both cards have bus-type ARCNET interfaces.

Commodore is currently developing AmigaDOS client software for Novell Netware and Amiga/NFS software including TCP/IP support for the Amiga series of computers. Details on Amiga/NFS are covered in a separate DevCon session. ♦



SANA: Standard Amiga Network Architecture

Preliminary Specification

by Dale Luck

Commodore-Amiga, Inc.

A network is a collection of services that allow the sharing of resources such as printers and disk drives by multiple users on a single computer, and the transmission of information such as electronic mail and files from one computer to another.

Most networks are designed as series of layers, each one using the services of the layer below it. Data coming from the application program, traverses the software layers until it reaches the physical hardware and is sent to the destination host's physical hardware. The data then rises through the layers on the host machine until it reaches the destination application. The reason for this layering is to provide a way of solving the myriad of problems that come up in a network environment. Some of these problems are:

- ☐ Connection failure due to hardware crash or OS crash
- ☐ Network traffic jams and congestion
- ☐ Packet (unit of information exchange) corruption, loss, delays, duplication, and sequence errors
- ☐ Bridging and routing of packets from a host on one network to a host on another network
- ☐ Multiple logical connections per transport mechanism
- ☐ Concurrent support of different types of network hardware
- ☐ Providing high level program and user oriented services
- ☐ Monitoring and maintainability

Each layer in the architecture provides solutions to one or more of these various problems. Higher level layers are able to depend on the lower layers for solutions to low-level problems and can then concentrate on providing solutions for higher level problems.

Existing Layering Models

The ISO-OSI 7 layering model is shown below. (International Standards Organization, Open Systems Interconnect)

Application
Presentation
Session
Transport
Network
Data Link
Hardware

Hardware Specifies electrical characteristics of the transport medium as well as the procedures used to arbitrate the use of the data carrying mechanism.

Data Link Specifies the format of the frames or packets that are sent on the hardware. How to recognize frame start and stop as well as some error detection for recognizing corrupt packets. It specifies an exchange of acknowledgments that allows two machines to know when a packet has been successfully transferred.

Network Specifies the destination addressing and routing. It breaks packets into smaller fragments that are too large for the Data Link layer. It takes care of network congestion problems.

Transport Provides end-to-end reliable connection between the destination and source hosts. Provides extra level of error detection and correction to make sure that no computer in the middle has failed.

Session Provide authentication, accounting, type of transmission mechanism, insuring that a group of network operations are not aborted when only partially complete.

Presentation A collection of routines for data transformations. Useful for data compression, translation, and dealing with different type of terminals.

Application Examples include electronic mail, remote device sharing, and file sharing.

Not all layers are used for every type of data transfer. Many applications talk directly to the transport layer and do their own connection authentication, data translations, and user interface management.

The Internet Layering Model:

Application
Transport
Internet
Network Interface
Hardware

Application Basically the OSI (session, presentation, and application) in one.

Transport Also similar to the ISO-Transport layer. However it further insures proper delivery by using acknowledgements and retransmission. It also fragments the data stream into pieces for lower layers.

Internet Error checking, routing.

Network Interface Accepts Internet datagrams formats them for the particular hardware and transmits them.

Hardware Same as the OSI model.

It will be apparent in the following discussion that layering is the appropriate method to use for the Amiga network architecture.

There are three main questions that when answered help describe a particular network:

- ☐ What type of hardware is used to transmit data between computers?
Ethernet, Arcnet, serial, parallel, FDDI, flashtalk, localtalk
- ☐ What protocol is used to send information between programs?
TCP/IP, IPX/SPX, DECnet, Tops, DDP, XNS
- ☐ What kinds of capabilities does the network provide?
File sharing, device sharing, electronic mail, remote session

The hardware connections can usually be summed this way: the higher the cost, the faster and more reliable the network. Connection types range from being built into every computer such as a standard serial interface at 19200 baud which is basically free, to plug boards that support the FDDI (Fiber Optic) standard at 100,000,000 baud which cost about \$4000 per node. Here is a table of example network hardware available today:

Hardware name	Speed bits/sec	Estimated cost per node	Access Control
Serial	19,200	\$15	None needed
LocalTalk	250,000	\$75	CSMA/CA
Arcnet	2,500,000	\$250	Token Bus
Ethernet	10,000,000	\$400	CSMA/CD
FDDI	100,000,000	\$4000	Token Ring

Protocols give programs the ability to communicate with each other without knowing the details of the physical hardware that transmits the data. Various protocols have become dominant in different markets and on specific vendors computers:

TCP/IP is the dominant protocol used on Unix machines.

DECnet is the dominant protocol used on DEC machines.

AppleTalk is the dominant protocol used by Apple Macs.

IPX/SPX (Novell) is one of the protocols used by IBM PC compatibles.

OSI is a recent protocol defined by ISO.

There are other protocols as well.

Each implementation of a protocol accepts data from a sending program, formats the data in a particular way for addressing and error recovery and sends it as a packet to the hardware. The receiving hardware accepts the packet of data and sends it to the protocol handler on the destination machine. The protocol handler decodes the specially formatted packet, extracting the original data that the sender transmitted, decodes the final destination address and sends the data to the intended receiving program.

Each protocol standard generally has a different way of doing this service since they were developed in different environments at different time periods and to take care of different needs. They differ on the number of computers they can talk to and how they handle errors.

Theoretically, each protocol is independent of the actual hardware that transmits the information. The types of services that are available to users varies a great deal depending on the network. Some support electronic mail (email) from one user to another user. Some allow sharing individual printers by more than one user or computer. Some allow backups of hard disks on different machines in the network to be done by a single tape drive on one of the computers in the network. Some allow file copying from one computer to another, and some allow transparent file access/sharing of entire file systems or directories from one main file server to several users on different computers.

These services are theoretically independent of the protocol used by the network.

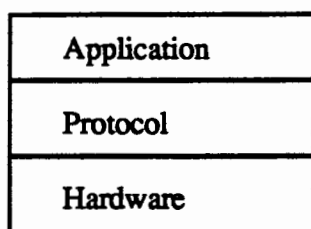
In the world as it exists today, the theoretically possible is not necessarily the available.

- ☐ Some protocols are not implemented on some computers.
- ☐ Some types of network hardware are not supported by some protocols.
- ☐ Most network services and applications do not support all popular protocols.
- ☐ Many networks only support one of the available protocols.
- ☐ Some applications are written using a particular programming interface to the network making it difficult to move that program to another machine that uses a similar function interface but with a different syntax.

Given all these possible compatibility problems how does one define a network architecture that will provide *interoperability*?

Definition of Interoperability

Interoperability is the ability of software and hardware on multiple machines from multiple vendors to communicate meaningfully. The answer is in defining an architecture that is hardware and protocol independent and that maintains a consistent and full function interface between all the hardware and protocol independent modules. If we take a look at dependencies we see that applications and services use protocols and protocols use hardware. A very simple layering model can be drawn that shows this:



The simple layering model gives the basic idea of dependencies however it implies that only a single application talks to a single protocol which interfaces to only a single piece of hardware at any one time.

What we want is to have multiple applications possibly running with multiple protocols over more than one type of network hardware. To have more than one protocol access either a single piece of hardware or have a single protocol access more than one type of hardware or any combination of the two, we cannot have the protocol layer interfacing directly to the hardware. There needs to be another layer insulating the two and presenting a set of device independent functions to the protocol software. This layer is the Data Link (ISO-OSI) or the Network Interface layer (Internet model). The job of this layer is to:

- ☐ Accept packets from one or more protocol sources on this computer and destined to another computer on this particular network hardware.
- ☐ Combine them into a stream of packets (*multiplexing*) and give them to the hardware to be sent to the destination computer.
- ☐ Accept packets from the hardware destined for this local computer and to any protocol.
- ☐ Separate packets based on packet type (*demultiplexing*) and send them to the appropriate protocol service
- ☐ Hide the details of this particular type of network hardware made by this particular vendor from the higher level layers.

Since the primary job of this layer is the multiplexing and demultiplexing of packets between the upper level layers and the lower level hardware layer, we call this Network Multiplexing Layer.

Application
Protocol
Network Multiplex
Hardware

Novell has additionally split the Network Multiplexor layer into two layers:

Protocol
<i>Link Support Layer</i>
<i>Multiple Link Interface Device</i>
Hardware

Quoting from *The Open Link Interface Specification: The Novell Position*:

"The Link Support Layer provides the link between a protocol stack and the appropriate adapter driver. It controls queues for outbound packets and routes received packets to the correct protocol stack."

The MLIDs are simple device interface drivers for a particular type of network interface hardware made by a particular vendor. The SANA NetMux layer can be seen as doing the job of both Novell's LSL and MLID since we feel that the LSL may need to be tied closer to the hardware as well as there may be a severe performance penalty for program control going through the additional layer. This position may change in the future given supporting evidence for the additional layer.

How is the NetMux layer going to be implemented on the Amiga?

- ☐ Using the Amiga device software model, the NetMux can be sent messages from more than one protocol stack and process them in sequential order. This is the nature of an Amiga device driver.
- ☐ The device driver is responsible for taking care of the hardware specific details, such as addressing and local buffering.
- ☐ The device driver will be programmed to accept packets destined for the local address, ignoring all other packets. The hardware itself may be able to accomplish this function. If not, it will have to be done by software itself.
- ☐ By requiring all packets to be self identifying, the device driver will use some type of identification function that will allow it to direct the packets to the proper protocol stacks. This identification function will be specified by the individual protocol stacks.

The Protocol Layer

Each network protocol stack will send packets to and receive packets from the NetMux. Since each NetMux will respond to the same commands, making the protocol work with different devices will be greatly simplified.

The network protocol stacks also interface to higher level layers, such as the application layer. The protocol stacks must be able to establish and maintain multiple logical connections over a single protocol space for one or more concurrent applications.

- ☐ Open and close multiple logical connections from different or the same processes
- ☐ Accept data from multiple processes
- ☐ Provide some control over the workings of the logical connection for the application. (buffering, etc.)
- ☐ Create protocol packets specifying final process destination based on protocol addressing model.
- ☐ Send packets to any required NetMux's
- ☐ Accept packets from any of the available NetMux's.
- ☐ Strip protocol dependent information and route the data to the intended process.

Since each protocol stack must support multiple process's concurrent access to this service, the Amiga device model is again used. The message based communication method conveniently hides most of the internal details of how this protocol stack does its work.

Network Interface Library

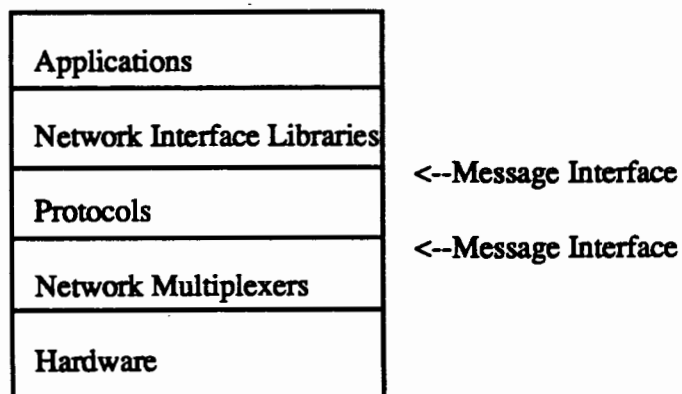
Although applications can be written to talk directly to the Protocol layer with the Amiga device commands and IOReq's, it is recommended that they be written to use one of the the industry standard network interface routines or libraries.

For example:

BSD sockets - used on most Unix machines
TLI - used by AT&T for SVR4
NetWare IPC - used by Novell

The Network Protocol Devices (NPDs) will provide all the capabilities required by the Network Interface Libraries. Because the NPDs are a separate layer, not part of the application interface routines, and designed to use the Amiga device software model, they will support more than one kind of network interface library concurrently. This will make it possible to run applications base on BSD sockets at the same time as applications based on AT&T's SVR4-TLI interface or applications written using the Novell interface libraries.

Now we have an additional conceptual layer that is actually part of the application, so our model now looks like this:



At present, SANA is an on going research and development project. All specifications are subject to change. Suggestions, comments, criticisms, and grammar corrections are welcome.

Additional Commodore-Amiga documents relating to SANA:

- ☐ Autodocs for the *ipc device* mechanism. These describe the current set of commands that the Protocol layer must support.
- ☐ Autodocs for the *ipcmem device*. These describe the current set of commands that the Network Multiplexor must support.

Suggested Reading

<i>Computer Networks</i>	Andrew S. Tanenbaum
<i>Internetworking with TCP/IP</i>	Douglas Comer
<i>Inside AppleTalk</i>	Gursharan Sidhu, Richard Andrews, and Alan Oppenheimer
<i>The Open Link Interface Specification: The Novell Position</i>	Novell, Inc.

TABLE OF CONTENTS

ipc.device
ipc.device/AbortIO
ipc.device/BeginIO
ipc.device/CloseDevice
ipc.device/CMD_CLEAR
ipc.device/CMD_FLUSH
ipc.device/CMD_INVALID
ipc.device/CMD_READ
ipc.device/CMD_RESET
ipc.device/CMD_START
ipc.device/CMD_STOP
ipc.device/CMD_UPDATE
ipc.device/CMD_WRITE
ipc.device/Expunge
ipc.device/IPC_CMD_ACCEPT
ipc.device/IPC_CMD_CONNECT
ipc.device/IPC_CMD_DCLOBJECT
ipc.device/IPC_CMD_DISCONNECT
ipc.device/IPC_CMD_FINDNODENAME
ipc.device/IPC_CMD_GETPEERNAME
ipc.device/IPC_CMD_QUERY
ipc.device/IPC_CMD_READMSG
ipc.device/IPC_CMD_SELECT
ipc.device/IPC_CMD_WRITEMSG
ipc.device/OpenDevice

ipc.device

ipc.device

ipc.device information

The ipc.device is a sample implementation of the Standard Amiga Network Architecture Protocol Device Layer. It supports the UNIXDOMAIN address resolution which is a filename on the the local system.

The SANA Protocol Device provides for the creation, maintainance, usage, and termination of bidirectional connections between two processes. This is a preliminary reference manual and represents work in progress. Many of the functions are tentatively defined and their actual defination may change in the future. This is to be used as draft reference guide for network developers.

TCP/IP, DECnet, AppleTalk, OSI, etc. would all be implemented in a module similar to this module.

This software module makes use of lower level device drivers known as the NetMux layer in SANA.

For more information you should get a copy of the SANA Theory of Operation. SANA = Standard Amiga Network Architecture

The following commands are required at this time to support the current AmigaDos BSD socket emulation library:

AbortIO,	BeginIO,	CloseDevice
Read,	Write,	Expunge
Accept,	AddObject,	Connect
GetPeerName,	Disconnect,	FindNodeName
Query,	Select,	OpenDevice
Invalid		

The following are unused at this time and may therefor be nops are unsupported. Their status may change though. At some time in future we will decide their status permanently.

Clear,	Flush,	WriteMsg
Reset,	Start,	Stop
Update,	ReadMsg	

Open Issues:

- Address translation and how it will work with hardware devices with varying address widths. Can it be made somewhat device independent?
- Objects are not clearly defined anywhere and you won't have a clue of what we mean by them unless you are already familiar with DECNET. An object is like an inet bound address. I don't know what it maps to in Novell land or AppleTalk.

ipc.device/AbortIO

ipc.device/AbortIO

NAME

AbortIO(ioRequest) -- abort an I/O request
A1

FUNCTION

This is an exec.library call.

This function aborts an ioRequest.
If the request is active, it is stopped immediately. If the request is
queued, it is painlessly removed. The request will be returned
in the same way as if it had normally completed.

INPUTS

ioRequest -- pointer to the IORequest Block that is to be aborted.

RESULTS

io_Error -- #IOERR_ABORTED (-2)

BUGS

NAME

BeginIO(ioRequest) -- start up an I/O process
A1

FUNCTION

This function initiates a I/O request made to the ipc device. This is a direct function call to the device. It gives the programmer more control over the io operation of the device. Most operations can usually be handled with exec's DoIO() and SendIO(). This function never blocks and control is returned to the caller.

INPUTS

ioRequest -- pointer to an I/O Request Block of size
io_ExtipcSize (see ipc.i for size/definition),
io_Message A mn_ReplyPort is required
io_Device set by OpenDevice
io_Unit set by OpenDevice
io_Command needs to be set to proper command
io_Flags If the IOB_QUICK bit is set the device will
attempt to finish command without 'Replying'
throughout the exec message system.
The IOB_QUICK bit will be cleared if the
request will 'Replied' when complete.
io_Length value depends on particular command
io_Data value depends on particular command

RESULTS

io_Error -- if the BeginIO succeeded, then Error will be null.
If the BeginIO failed, then the Error will be non-zero.
I/O errors won't be reported until the io completes.

SEE ALSO

devices/ipc.h

ipc.device/CloseDevice

ipc.device/CloseDevice

NAME

CloseDevice -- close the ipc device

SYNOPSIS

CloseDevice(ioRequest)
A1

FUNCTION

This is an exec call that terminates communication with the
ipc device.

INPUTS

ioRequest - pointer to ioRequest block initialized by OpenDevice

SEE ALSO

ipc.device/OpenDevice

ipc.device/CMD_CLEAR

ipc.device/CMD_CLEAR

NAME

Clear -- clear internal ipc read buffers.

FUNCTION

This command causes device to dump all pending unread data on the floor and set bytes available for reading to 0. All available data for reading is lost.

IO REQUEST

io_Command CMD_CLEAR

RESULTS

Error -- if the Clear succeeded, then io_Error will be null.
If the Clear failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

Not implemented and not known to be useful.

ipc.device/CMD_FLUSH

ipc.device/CMD_FLUSH

NAME

Flush -- clear internal ipc write buffers.

FUNCTION

This command causes device to dump all pending unwritten data on the floor. All data is lost. Abort all I/O requests. Flush will not affect IoRequests actually in progress.

IO REQUEST

io_Command CMD_FLUSH

RESULTS

Error -- if the Flush succeeded, then io_Error will be null.
If the Flush failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

Not implemented and not known to be useful at this time.

ipc.device/CMD_INVALID

ipc.device/CMD_INVALID

NAME

Invalid -- Return with error IOERR_NOCMD

FUNCTION

This command causes device driver to reply with an error IOERR_NOCMD as defined in exec/errors.h indicating the command is not supported.

IO REQUEST

io_Command CMD_INVALID

RESULTS

Error -- io_Error is filled with IOERR_NOCMD

SEE ALSO

BUGS

ipc.device/CMD_READ

ipc.device/CMD_READ

NAME

Read -- Read input from ipc connection

FUNCTION

This command causes the protocol device to return with a buffer of data received from the process on the other end of the virtual connection.

The Query function can be used to check how many characters are currently waiting in the ipc buffer.

This command only guarantees that at most io_Length bytes will be read. The actual number of bytes read will be returned in io_Actual.

If there are no bytes available to be read at the time the request is made, the request will not be replied to until some data has arrived.

The ipcFlags has a special Flag that can be set that will cause the device to never block the request even if there are no bytes available for reading. In this case io_Actual will be set to zero (0) and io_Error will be set to EWOULDBLOCK. This flag is called ipc_NOWAITIO.

IO REQUEST

io_Command	CMD_READ
io_Length	max number of characters to receive.
io_Data	pointer to buffer

RESULTS

io_Actual -- number of received bytes
Error -- if the Read succeeded, then io_Error will be 0.
If the Read failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

ipc.device/IPCMD_QUERY
ipc.device/CMD_WRITE

BUGS

ipc.device/CMD_RESET

ipc.device/CMD_RESET

NAME

Reset -- Reinitialize ipc protocol state machine

FUNCTION

This function causes the ipc device to abort all I/O requests both queued and in process. It will close all ipc connections and merrily throw all data coming in and data ready to go out on the floor. Do not use this function lightly. It is intended for testing purposes and for super user applications. software that is managing the functions of the entire machine.

IO REQUEST

io_Command CMD_RESET

RESULTS

Error -- if the Reset succeeded, then Error will be null.
If the Reset failed, then the Error will be non-zero.

BUGS

It is not clear that this command should be a standard command at this layer of the network protocol.
This command is presently not implemented.

ipc.device/CMD_START

ipc.device/CMD_START

NAME

Start -- Restart all I/O on this ipc connection

FUNCTION

This command restarts I/O on this connection.
I/O can be paused by a CMD_STOP command.

IO REQUEST

io_Command CMD_Start

RESULTS

Error -- if the Start succeeded, then io_Error will be null.
If the Start failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection
shutdown or network termination.

SEE ALSO

BUGS

It is not clear whether this function is that useful since
this layer of the ipc protocol guarantees proper delivery
of data and does its own flow control.

ipc.device/CMD_STOP

ipc.device/CMD_STOP

NAME

Stop -- Pause all I/O on this ipc connection

FUNCTION

This command halts all current I/O on this connection.
I/O can be resumed by a CMD_START command.

IO REQUEST

io_Command CMD_STOP

RESULTS

Error -- if the Stop succeeded, then io_Error will be null.
If the Stop failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection
shutdown or network termination.

SEE ALSO

BUGS

It is not clear whether this function is that useful since
this layer of the ipc protocol guarantees proper delivery
of data and does its own flow control.

ipc.device/CMD_UPDATE

ipc.device/CMD_UPDATE

NAME

Update -- Force all write buffers out to connection.

FUNCTION

This command causes device to send any data waiting in the internal cache to be written to the actual ipc hardware.

This is used when the upper protocol needs to guarantee that every piece of data has actually been sent to the other side. Useful for terminal emulators which do not want any buffering done by low level software.

IO REQUEST

io_Command CMD_UPDATE

RESULTS

Error -- if the Update succeeded, then io_Error will be null.

 If the Update failed, then io_Error will be non-zero.

 io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipc.device/CMD_WRITE

ipc.device/CMD_WRITE

NAME

Write -- Write output to ipc connection

FUNCTION

This command causes data to be sent to a process on the other end of the ipc connection.

If there is more data being sent by this command than can be accepted by the connection immediately then block the request until all the data has been sent.

The ipcFlags has a special flag that can be set that will cause the device to never block the request. In this case then as much data as can be sent will be sent and io_actual will contain the number of bytes actual sent. If no bytes were written, the IO_Error will be set to EWOULDBLOCK. This flag is called ipc_NOWAITIO.

IO REQUEST

io_Command	CMD_WRITE
io_Length	number of characters to write
io_Data	pointer to buffer

RESULTS

io_Actual -- number of bytes sent
Error -- if the Write succeeded, then io_Error will be null.
If the Write failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipc.device/Expunge

ipc.device/Expunge

NAME

Expunge -- Free all system resources and dependencies

FUNCTION

This function deallocates all memory and functionality associated with the ipc device. This includes the data section for the ipc device, the break-related message ports, read and write queues and buffers, and interrupt vector attachments. If the device is currently closed, Expunge takes place immediately. If it is Open, the Expunge cannot take place.

RESULTS

Error -- if the Expunge succeeded, then Error will be null.
If the Expunge failed, then the Error will be non-zero.

ipc.device/IPCMD_ACCEPT

ipc.device/IPCMD_ACCEPT

NAME

Accept -- Accept incoming connection at Object

FUNCTION

This command causes device driver to accept a connection request at ipc port Object and establish a bidirectional communication path to the process requesting the connection. The IOReq used must be the same one that was returned by the IPCMD_DCLOBJECT command. This information is the same information one gets if a CMD_GETPEERNAME is used.

After successful completion of the ACCEPT command this IOReq can be used for QUERY, READ, WRITES, etc. on this virtual connection.

IO REQUEST

io_Command IPCMD_ACCEPT
io_Data Points to a buffer for name of peer
io_Length Size of buffer

RESULTS

Error -- if the Accept succeeded, then io_Error will be null.
If the Accept failed, then io_Error will be non-zero.
io_Error will indicate problems such as network down. If there is no pending connection an error will result.
IO-Actual - number of bytes transmitted to buffer.

SEE ALSO

ipcDevice/IPCMD_CONNECT
ipcDevice/IPCMD_ADDOBJECT

BUGS

ipc.device/IPCMD_CONNECT

ipc.device/IPCMD_CONNECT

NAME

Connect -- Attempt to connect to ipc port

FUNCTION

This command causes the device driver to attempt to establish a bidirectional connection to an Object that is waiting for connections.

This command will wait until either a valid connection has been established or some sort of error has occurred. The new flags for NOWAITIO are not supported. This command always blocks. The programmer may AbortIO this command if they need to time it out.

IO REQUEST

io_Command IPCMD_CONNECT

RESULTS

Error -- if the Connect succeeded, then io_Error will be null.
If the Connect failed, then io_Error will be non-zero.
io_Error will indicate problems such as Object not found or network down.

SEE ALSO

ipcDevice/IPCMD_ACCEPT

BUGS

ipc.device/IPCMD_DCLOBJECT

ipc.device/IPCMD_DCLOBJECT

NAME

DclObject -- Declare an Object for accepting a single connection

FUNCTION

This command causes device driver to make this Object available as a connection point for incoming connections.

An Object is a single entity that once a connection is made it is removed from the table of available connections.

A task can create multiple Objects with the same name and each one in turn will be used up as connections are made.

The request is not completed until:

- o A connection attempt is started.
- o An error has occurred.
- o The request is aborted.

IO REQUEST

io_Command	IPCMD_DCLOBJECT
io_Data	pointer to buffer describing object
io_Length	size of buffer

RESULTS

Error -- if the DclObject succeeded, then io_Error will be null.

If the DclObject failed, then io_Error will be non-zero.

io_Error will indicate problems such as duplicate Object or network down.

SEE ALSO

ipcDevice/IPCMD_ACCEPT
ipcDevice/IPCMD_CONNECT

BUGS

For device ipc.device:

If the two name strings are different but really reference the same file, the connection will not be made. Also if two processes in different directories use the same name relative to their current working directory, they will be connected.

ipc.device/IPCMD_DISCONNECT

ipc.device/IPCMD_DISCONNECT

NAME

Disconnect -- Disconnect from ipc connection

FUNCTION

This command causes device driver to disconnect from an ipc connection.

Any data that is waiting to be read by this process will be discarded. Any data that has been written will be sent to the waiting process on the other end of this connection. To completely terminate a connection, both sides must do a disconnect. It does not matter who terminates the connection first.

IO REQUEST

io_Command IPCMD_DISCONNECT

RESULTS

Error -- no error is possible with this command.

SEE ALSO

ipcDevice/IPCMD_ACCEPT
ipcDevice/IPCMD_CONNECT

BUGS

ipc.device/IPCMD_FINDNODENAME

ipc.device/IPCMD_FINDNODENAME

NAME

FindNodeName -- Translate Node address to name and vice versa.

FUNCTION

Given the address, look up the corresponding name of the node in the connection database.

Given the name look up the corresponding address in the connection database.

IO REQUEST

io_Command IPCMD_FINDNODENAME

RESULTS

Error -- if the FindNodeName succeeded, then io_Error will be null.

If the FindNodeName failed, then io_Error will be non-zero.

io_Error will indicate problems such Node not found.

SEE ALSO

BUGS

It may be more clear to have separate commands for:

FindNodeNameFromAddress

FindAddressFromNodeName

ipc.device/IPCMD_GETPEERNAME

ipc.device/IPCMD_GETPEERNAME

NAME

GetPeerName -- Get the name of the peer on this connection

FUNCTION

This command returns the name of host that the connection is communicating with.

IO REQUEST

io_Command	IPCMD_GETPEERNAME
io_Data	ptr to buffer to store name
io_Length	size of the buffer

RESULTS

io_Actual -- number of bytes in the name of the peer
Error -- if the Query succeeded, then io_Error will be null.
If the Query failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.
If io_Data is too small io_Error will be set to EFAULT

SEE ALSO

BUGS

ipc.device/IPCMD_INTERNAL

ipc.device/IPCMD_INTERNAL

NAME

Internal -- Internal status of ipc connection

FUNCTION

This command is a hook for network developers for getting access to the internals of their network device. The parameters are completely dependant on the implementation and no attempt at this time is being made to standardize anything.

For the ipc.evice subsystem, CMD_INTERNAL presently fills in a buffer pointed to by io_Data with copies of pointers to struct IpcInternal

```
{
    struct List *Objects;
    struct List *ActiveObjects;
    struct List *PendingSelects;
    struct SignalSemaphore *ss;
};
```

In the interest of leverage and efficiency, a general network debugging and monitoring program could possibly be written if the developers could standardize on some of the internal data structures, but that is usually asking too much. ;-)

IO REQUEST

```
io_Command IPCMD_INTERNAL
io_Data      points to buffer to put data
io_Length    size of buffer
```

RESULTS

```
io_Actual -- number of bytes transferred
Error -- if the Internal command is not supported return
        CMDINVALID.
        Or if the buffer is not big enough.
```

SEE ALSO

BUGS

ipc.device/IPCMD_QUERY

ipc.device/IPCMD_QUERY

NAME

Query -- Query status of ipc connection

FUNCTION

This command causes device driver to report the status of the ipc connection. The number of unread bytes in the input buffer of the connection are in io_Actual. The number of logical messages are also noted somewhere. The number of bytes in the first message is also available.

IO REQUEST

io_Command IPCMD_QUERY

RESULTS

io_Actual -- number of bytes available for reading
Error -- if the Query succeeded, then io_Error will be null.
 If the Query failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

Current implementation only supports:
Total bytes readable.

ipc.device/IPCMD_READMSG

ipc.device/IPCMD_READMSG

NAME

ReadMsg -- Read complete message from ipc connection

FUNCTION

This is a alternate interface to the read mechanism of the virtual circuit and provides a datagram method of delivery. This command causes device driver to return with a buffer of data received from the process on the other end of the connection as sent in a single WriteMsg.

The Query function can be used to check how many messages are currently waiting in the ipc buffer, how big the first message is and how many total bytes for all the messages are available.

This command only guarantees that at most io_Length bytes will be read. The actual number of bytes read will be returned in io_Actual.

Also if there are more than one message available and io_Length could handle both, this command will only read a single message of data.

If the first message available to be read has more bytes than io_Length, io_Error will be set to IOERR_MESSAGE_TOO_BIG, the request will be sent back, and the message will be left in the input queue of the ipc connection. The size of the message can be gotten with CMD_QUERY.

If there are no messages available to be read at the time the request is made, the request will not be replied to until a message has arrived.

The ipcFlags has a special Flag that can be set that will cause the device to never block the request even if there are no messages available for reading. In this case io_Actual will be set to zero (0).

This flag is called ipc_NOWAITIO.

IO REQUEST

io_Command	IPCMD_READMSG
io_Length	max number of characters to receive.
io_Data	pointer to buffer

RESULTS

io_Actual -- number of received bytes
Error -- if the ReadMsg succeeded, then io_Error will be 0.
If the ReadMsg failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination or Message too big.

SEE ALSO

ipc.device/IPCMD_QUERY
ipc.device/IPCMD_WRITEMSG

BUGS

NAME

Select -- Select status of ipc connection

FUNCTION

The Select command causes the network device to examine a list of supplied IOReq's and finish the command if one or more conditions are present on those connections. If none of the conditions are satisfied the IOReq will be deferred. When one or more of the conditions is satisfied then the IOReq will be finished.

Conditions to be satisfied:

- Message ready for reading?
- Data available for reading?
- Space available to write data?
- Some sort of exception? (Broken connection, etc.)

In each of the IOReq's, set the bits in SelectRequest for the conditions requested.

When the IOReq's return, examine the corresponding bits in the SelectResults to find out what caused the completion of the select command for that IOReq. There may be more than one bits set.

This command is never completed via QUICKIO it always does a ReplyMsg.

Setting the User.Flags bit ipcNOWAITIO will cause the device to finish the command even if there are no terminating conditions present. The SelectResults fields of the IOReqs will be set to zero if none of the terminating conditions for that connection were present. This is useful for polling a set of connections to see if any are ready or to see if a particular connection is still alive.

If the connection referenced by an IOReq is going down, the IO.io_Error field for that IOReq will be set to nonzero.

It is illegal to have more than one IOReq on the list that point to the same active connection.

IO REQUEST

io_Command	IPC_CMD_SELECT
io_Data	points to a list of IOReq;
SelectRequests	For every IOReq in the list set the bits for the conditions for this connection.
SelectResults	Must be initialized to 0

RESULTS

SelectResults	- treated as a mask. Bits corresponding to input available or output ready will be set.
io_Error	- always return null unless Aborted.
	io_Error in the individual IOReq's will contain an Error code for exceptions.

SEE ALSO

BUGS

ipc.device/IPCMD_WRITEMSG

ipc.device/IPCMD_WRITEMSG

NAME

WriteMsg -- Write complete message to ipc connection

FUNCTION

This command causes device driver to send a message of io_Length bytes to the process on the other end of the connection. This is a datagram interface to the virtual connection.

If the message cannot be sent immediately because there are not any buffers for this message then block the request until the message can be sent.

The ipcFlags has a special Flag that can be set that will cause the device to never block the request even if the message cannot be accepted immediately. In this case io_Actual will be set to zero (0). This flag is called ipcNOWAITIO.

IO REQUEST

io_Command	IPCMD_WRITEMSG
io_Length	number of characters in message to send.
io_Data	pointer to buffer

RESULTS

io_Actual -- number of bytes sent
Error -- if the WriteMsg succeeded, then io_Error will be null.
If the WriteMsg failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination

SEE ALSO

ipc.device/IPCMD_QUERY
ipc.device/IPCMD_WRITEMSG

BUGS

NAME

OpenDevice -- Request an opening of the ipc device.

SYNOPSIS

```
error = OpenDevice(protocol_name, unit, ioRequest, flags)
D0                A0                D0    A1                D1
```

```
BYTE OpenDevice(STRPTR, ULONG, struct IOStdIpc *, ULONG);
```

FUNCTION

This is an exec call. Exec will search for the device named by protocol_name, and if found, will pass this call on to the device.
For the memory protocol device the current name is "ipc.device".
The ioRequest is initialized for use with BeginIO. All IORequests must be initialized with this call.

This structure is found in ipc.h

```
struct IOStdIpc {
    struct IOStdReq IO; /* standard Amiga i/o request */
    struct {
        unsigned charSelectRequests;
        unsigned charSelectResults;
        struct Node    SelectNode;
        unsigned short  Flags;
        void *read;     /* internal use pointer */
        void *write;    /* internal use pointer */
    } User; /* IOStdReq extension for ipcnet */
};
```

INPUTS

```
protocol_name  - pointer to literal string "ipc.device"
unit           - Must be zero
ioRequest      - pointer to an ioRequest block of size IOStdReq
                  to be initialized by the ipc.device.
                  (see devices/ipc.h for the definition)
flags          - Must be zero for future compatibility
```

RESULTS

```
D0            - same as io_Error
io_Error      - If the Open succeeded, then io_Error will be null.
                  If the Open failed, then io_Error will be non-zero.
io_Device     - A pointer to whatever device will handle the calls
                  for this unit. This pointer may be different depending
                  on what unit is requested.
```

BUGS

SEE ALSO

ipc.device/CloseDevice

TABLE OF CONTENTS

ipcmem.device
ipcmem.device/AbortIO
ipcmem.device/BeginIO
ipcmem.device/CloseDevice
ipcmem.device/CMD_BROADCAST
ipcmem.device/CMD_CLEAR
ipcmem.device/CMD_FLUSH
ipcmem.device/CMD_INVALID
ipcmem.device/CMD_READ
ipcmem.device/CMD_RESET
ipcmem.device/CMD_START
ipcmem.device/CMD_STOP
ipcmem.device/CMD_UPDATE
ipcmem.device/CMD_WRITE
ipcmem.device/Expunge
ipcmem.device/IPCMD_ADDMULTICASTADDRESS
ipcmem.device/IPCMD_CONNECT
ipcmem.device/IPCMD_DELMULTICASTADDRESS
ipcmem.device/IPCMD_DISCONNECT
ipcmem.device/IPCMD_GETADDRESS
ipcmem.device/IPCMD_GETSTATS
ipcmem.device/IPCMD_QUERY
ipcmem.device/IPCMD_READANDCLEARSTATS
ipcmem.device/IPCMD_SETADDRESS
ipcmem.device/OpenDevice

ipcmem.device

ipcmem.device

ipcmem.device information

The ipcmem.device is a sample implementation of the Standard Amiga Network Architecture Network Multiplexor Device Layer.

The SANA NetMux Device Layer provides the services for accessing the physical network hardware, in this case memory. It controls the multiplexing of packets going out on the hardware and directs incoming packets from the hardware to the proper higher level handler.

Software that directly controls ethernet hardware, Arcnet hardware, FDDI, etc. should support the same commands as the ipcmem.device.

This is a preliminary document and describes work in progress. Many of the functions are tentatively defined and their actual definition may change in the future. This is to be used as draft reference guide for network developers.

For more information you should get a copy of the SANA Theory of Operation. SANA = Standard Amiga Network Architecture

At present there is no known example implementation of this device although one is actually in progress and several more are in the planning stages.

Open Issues:

ipcmem.device/AbortIO

ipcmem.device/AbortIO

NAME

AbortIO(ioRequest) -- abort an I/O request
A1

FUNCTION

This is an exec.library call.

This function aborts an ioRequest.

If the request is active, it is stopped immediately. If the request is queued it is removed. The request will be returned in the same way as if it had normally completed.

INPUTS

ioRequest -- pointer to the IORequest Block that is to be aborted.

RESULTS

io_Error -- #IOERR_ABORTED (-2)

BUGS

NAME

BeginIO(ioRequest) -- start up an I/O process
A1

FUNCTION

This is a direct function call to the device. It is intended for more advanced programmers. See exec's DoIO() and SendIO() for the normal method of calling devices.

This function initiates a I/O request made to the ipc device.

INPUTS

ioRequest -- pointer to an I/O Request Block of size
io_ExtipcmemSize (see ipcmem.i for size/definition),
io_Message A mn_ReplyPort is required
io_Device set by OpenDevice
io_Unit set by OpenDevice
io_Command needs to be set to proper command
io_Flags If the IOB_QUICK bit is set the device will
attempt to finish command without 'Replying'
through the exec message system.
The IOB_QUICK bit will be cleared if the
request will 'Replied' when complete.
io_Length value depends on particular command
io_Data value depends on particular command

RESULTS

io_Error -- If the command is a not supported then
io_Error will be set to IOERR_NOCMD
otherwise io_Error is set by the command
when the command completes.

SEE ALSO

devices/ipcmem.h

ipcmem.device/CloseDevice

ipcmem.device/CloseDevice

NAME

CloseDevice -- close the ipc device

SYNOPSIS

CloseDevice(ioRequest)
 A1

FUNCTION

This is an exec call that terminates communication with the
ipc device. Upon closing, the device's input buffers are freed.

Note that all IORequests MUST be complete before closing.
If any are pending, your program must AbortIO() then WaitIO()
to complete them.

INPUTS

ioRequest - pointer to ioRequest block initialized by OpenDevice

SEE ALSO

ipcmem.device/OpenDevice

ipcmem.device/CMD_BROADCAST

ipcmem.device/CMD_BROADCAST

NAME

Broadcast -- Broadcast packet on ipc device

FUNCTION

This command causes the packet to be sent to all nodes on this device via a broadcast mechanism. The actual broadcast mechanism is device specific.

IO REQUEST

io_Command	CMD_BROADCAST
io_Length	number of characters to write
io_Data	pointer to buffer

RESULTS

io_Actual -- number of bytes sent

Error -- if the Broadcast succeeded, then io_Error will be null.

If the Broadcast failed, then io_Error will be non-zero.

io_Error will indicate problems such as connection shutdown or network termination. The broadcast may also fail if the network hardware does not support a broadcast mechanism and the software is unable to emulate one.

SEE ALSO

BUGS

ipcmem.device/CMD_CLEAR

ipcmem.device/CMD_CLEAR

NAME

Clear -- clear internal device read buffers.

FUNCTION

This command causes device to dump all pending unread data packets on the floor.
All available data for reading is lost.

IO REQUEST

io_Command CMD_CLEAR

RESULTS

Error -- if the Clear succeeded, then io_Error will be null.
If the Clear failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

Unimplemented and not known to be useful at this time.

ipcmem.device/CMD_FLUSH

ipcmem.device/CMD_FLUSH

NAME

Flush -- clear internal ipc write buffers.

FUNCTION

This command causes device to dump all pending unwritten packets on the floor. All data is lost. Abort all I/O requests. Flush will not affect IoRequests actually in progress.

IO REQUEST

io_Command CMD_FLUSH

RESULTS

Error -- if the Flush succeeded, then io_Error will be null.
If the Flush failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

Unimplemented and not known to be useful at this time.

ipcmem.device/CMD_INVALID

ipcmem.device/CMD_INVALID

NAME

Invalid -- Return with error IOERR_NOCMD

FUNCTION

This command causes device driver to reply with an error IOERR_NOCMD as defined in exec/errors.h indicating the command is not supported.

IO REQUEST

io_Command CMD_INVALID

RESULTS

Error -- io_Error is filled with IOERR_NOCMD

SEE ALSO

BUGS

ipcmem.device/CMD_READ

ipcmem.device/CMD_READ

NAME

Read -- Invalid command for ipc-device

FUNCTION

This command is not supported since all input must first go through a filter. To see how to get data from the device check out CMD_CONNECT. You will find that you are called with a packet ptr and must return in a short amount of time after deciding whether to keep and copy the packet somewhere else or just return with 0 rejecting the packet.

IO REQUEST

io_Command CMD_READ

RESULTS

io_Actual -- 0
Error -- will be ERR_CMDNOTSUPPORTED

SEE ALSO

ipcmem.device/CMD_CONNECT

BUGS

ipcmem.device/CMD_RESET

ipcmem.device/CMD_RESET

NAME

Reset -- Reset the network device to initialized state

FUNCTION

This function causes the device to abort all I/O requests both queued and in process.

Do not use this function lightly. It is intended for testing purposes and for super user applications and software that is managing the functions of the entire machine.

IO REQUEST

io_Command CMD_RESET

RESULTS

Error -- if the Reset succeeded, then Error will be null.

 If the Reset failed, then the Error will be non-zero.

BUGS

It is not clear that this command should be a standard command at this layer of the network protocol.

ipcmem.device/CMD_START

ipcmem.device/CMD_START

NAME

Start -- Restart all I/O on this ipc device

FUNCTION

This command restarts I/O on this device.
I/O can be paused by a CMD_STOP.

IO REQUEST

io_Command CMD_Start

RESULTS

Error -- if the Start succeeded, then io_Error will be null.
If the Start failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection
shutdown or network termination.

SEE ALSO

BUGS

Not implemented and not know to be useful at this time.

ipcmem.device/CMD_STOP

ipcmem.device/CMD_STOP

NAME

Stop -- Pause all I/O on this ipc device

FUNCTION

This command halts all current I/O on this device.
I/O can be resumed by a CMD_START command.

IO REQUEST

io_Command CMD_STOP

RESULTS

Error -- if the Stop succeeded, then io_Error will be null.
If the Stop failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection
shutdown or network termination.

SEE ALSO

BUGS

Not implemented and not known to be useful at this time.

ipcmem.device/CMD_UPDATE

ipcmem.device/CMD_UPDATE

NAME

Update -- Force all packets out to device

FUNCTION

This command causes the device to send any data waiting in the internal cache to be written to the actual ipc hardware. Use this command when higher level software needs to insure data is sent to the device immediately without waiting for additional bytes to put in a larger packet.

IO REQUEST

io_Command CMD_UPDATE

RESULTS

Error -- if the Update succeeded, then io_Error will be null.
If the Update failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipcmem.device/CMD_WRITE

ipcmem.device/CMD_WRITE

NAME

Write -- Write packet to network device

FUNCTION

This command causes the packet to be sent to a node at the specified address.

If there is more data being sent by this command than can be accepted by the device immediately then block the request until all the data has been sent.

There may be a maximum packet size acceptable by this device. To get that size use the cmd_Query function. cmd_Write will return an error if io_Length exceeds the maximum packet size allowed for this device.

IO REQUEST

io_Command	CMD_WRITE
io_Length	number of characters to write
io_Data	pointer to buffer

RESULTS

io_Actual -- number of bytes sent

Error -- if the Write succeeded, then io_Error will be null.

If the Write failed, then io_Error will be non-zero.

io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

NAME

Expunge -- Free all system resources and dependencies

FUNCTION

This function deallocates all memory and functionality associated with the ipc device. This includes the data section for the ipc device, the break-related message ports, read and write queues and buffers, and interrupt vector attachments. If the device is currently closed, Expunge takes place immediately. If it is Open, the Expunge cannot take place.

RESULTS

Error -- if the Expunge succeeded, then Error will be null.
If the Expunge failed, then the Error will be non-zero.

ipcmem.device/IPCMD_ADDMULTICASTADDRESS

NAME

AddMulticastAddress -- Add the multicast address for this ioreq

FUNCTION

This command causes the device driver to enable Multicast packet reception for the requested address.

IO REQUEST

io_Command IPCMD_ADDMULTICASTADDRESS
io_Data Pointer to multicast address
io_Length Size of buffer

RESULTS

Error -- if the AddMulticastAddress succeeded, then io_Error will be null. If the AddMulticastAddress failed, then io_Error will be non-zero. io_Error will indicate problems such as Multicast not supported.

SEE ALSO

DelMulticastAddress

BUGS

ipcmem.device/IPCMD_CONNECT

ipcmem.device/IPCMD_CONNECT

NAME

Connect -- Connect to physical link

FUNCTION

This command causes the device driver to physically connect to the network and begin receiving and sending packets.

The caller must supply a PacketType argument.

The incoming packets are checked against this protocol type and if they match are sent to the user.

Valid PacketType arguments have been arbitrarily set to the standard EthernetPacket types.

ETHERTYPE_NS	0x0600	/* XNS protocol */
ETHERTYPE_IP	0x0800	/* IP protocol */
ETHERTYPE_ARP	0x0806	/* Addr. resolution protocol */
ETHERTYPE_DN	0x6003	/* DECnet protocol */
ETHERTYPE_LAT	0x6004	/* LAT protocol */
ETHERTYPE_ATALK	0x809B	/* Appletalk */
ETHERTYPE_AARP	0x80F3	/* Appletalk Arp */
ETHERTYPE_RARP	0x8035	/* Reverse Arp */
ETHERTYPE_ALL	0x0000	/* All Packets (Amiga) */

Non ethernet based devices will accept the same PacketType arguments and must translate it to the proper PacketType for their device.

The IO_CMD Lookup_PacketType will let the user have access to the corresponding PacketType for this device.

There is a required second argument which must be set to zero.

At any one time there can be only one listener per PacketType. If another user already has connected using the requested PacketType an error will be returned with this new connect request.

You must Disconnect for every Connection request.

IO REQUEST

io_Command IPCMD_CONNECT

RESULTS

Error -- if the Connect succeeded, then io_Error will be null.

 If the Connect failed, then io_Error will be non-zero.

 io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipcmem.device/IPCMD_DELMULTICASTADDRESS

NAME

DelMultiCastAddress -- Delete the multicast address for this node

FUNCTION

This command causes device driver to disable MultiCast packet reception for the requested address.

IO REQUEST

io_Command IPCMD_DELMULTICASTADDRESS
io_Data Pointer to multicast addresses
io_Length Size of buffer

RESULTS

Error -- if the DelMultiCastAddress succeeded, then io_Error will be null. If the DelMultiCastAddress failed, then io_Error will be non-zero. io_Error will indicate problems such as Multicast not supported.

SEE ALSO

AddMulticastAddress

BUGS

ipcmem.device/IPCMD_DISCONNECT

ipcmem.device/IPCMD_DISCONNECT

NAME

Disconnect -- Disconnect from physical link

FUNCTION

This command releases the PacketType packets from being sent to this connection and makes them available for other connections. If there are no other listeners than the packets will be thrown away.

Decrement the local connect counter. If the connect counter reaches zero then really disconnect from the physical device. Programmer must disconnect from the device before closing the device.

IO REQUEST

io_Command IPCMD_DISCONNECT

RESULTS

Error -- if the Disconnect succeeded, then io_Error will be null.
If the Disconnect failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipcmem.device/IPCMD_GETADDRESS

ipcmem.device/IPCMD_GETADDRESS

NAME

GetAddress -- Get the local address of this node on this network

FUNCTION

This command causes device driver to copy the local node address into the buffer pointed to by io_Data;

IO REQUEST

io_Command IPCMD_GETADDRESS
io_Data Pointer to local node address
io_Length Size of buffer

RESULTS

Error -- if the GetAddress succeeded, then io_Error will be null.
If the GetAddress failed, then io_Error will be non-zero.
io_Error will indicate problems such as address not known.

SEE ALSO

BUGS

ipcmem.device/IPCMD_GETSTATS

ipcmem.device/IPCMD_GETSTATS

NAME

GetStats -- Get accumulated statistics for this device

FUNCTION

This command causes device driver to retrieve various runtime statistics for this device.

The data will be copied into a buffer pointed to by io_Data.

At most io_Length data will be copied.

Use cmd_QUERY to find out the size of the statistics structure for this device.

The data returned corresponds to the following structure:

```
struct IpcDeviceStats
{
    int packets_received;
    int packets_sent;
    int packets_waiting_in;
    int packets_waiting_out;
    int errors;
    int Connections;
    /* More to come */
};
```

IO REQUEST

```
io_Command      IPCMD_GETSTATS
io_Data         Pointer to buffer to copy statistics
io_Length       Length of buffer
```

RESULTS

io_Actual -- number of bytes available for reading

Error -- if the GetStats succeeded, then io_Error will be null.

If the GetStats failed, then io_Error will be non-zero.

io_Error will indicate problems such as connection shutdown or network termination.

SEE ALSO

BUGS

ipcmem.device/IPCMD_QUERY

ipcmem.device/IPCMD_QUERY

NAME

Query -- Query status of ipc device

FUNCTION

This command causes device driver to report the status and retrieve various information about the device. The information is copied into a buffer pointed to by io_Data. The format of the data is as follows:

```
#define MAX_ADDRESS_SIZE      8
struct IpcDevQuery
{
    unsigned char[MAX_ADDRESS_SIZE]    local_address;
    int address_size; /* number of bits */
    /*of local_address that are significant */
    int MaxPacketSize;
    /* More to come */
};
/* example 3=scsi, 8=arcnet, 1= serial, 48=ethernet */
```

IO REQUEST

io_Command IPCMD_QUERY
io_Data Pointer to buffer to copy information
io_Length Size of buffer

RESULTS

io_Actual -- number of bytes available for reading
Error -- if the Query succeeded, then io_Error will be null.
If the Query failed, then io_Error will be non-zero.
io_Error will indicate problems such as connection
shutdown or network termination.

SEE ALSO

BUGS

ipcmem.device/IPCMD_READANDCLEARSTATS

NAME

ReadAndClearStats -- Read and Reset statistics counters of ipc device

FUNCTION

First copy the current values of the statistics counters to the buffer. Then before they have a chance to be change, set them to zero.

IO REQUEST

io_Command IPCMD_RESETSTATS

RESULTS

io_Actual -- number of bytes in stat counter buffer

Error -- if the ReadAndResetStats succeeded, then io_Error will be null.

 If the ReadAndResetStats failed, then io_Error will be non-zero.

SEE ALSO

BUGS

There is still controversy over the defination of this function.

It's final defination will await more input from the network implementors and consultants.

ipcmem.device/IPCMD_SETADDRESS

ipcmem.device/IPCMD_SETADDRESS

NAME

SetAddress -- Set the local address of this node on this network

FUNCTION

This command causes device driver to set the local node address to the address pointed to by io_Data;

IO REQUEST

io_Command IPCMD_SETADDRESS
io_Data Pointer to local node address
io_Length Size of buffer

RESULTS

Error -- if the SetAddress succeeded, then io_Error will be null.
If the SetAddress failed, then io_Error will be non-zero.
io_Error will indicate problems such as address not settable.

SEE ALSO

BUGS

ipcmem.device/OpenDevice

ipcmem.device/OpenDevice

NAME

OpenDevice -- Request an opening of the network device.

SYNOPSIS

```
error = OpenDevice(network_device, unit, ioRequest, flags)
D0          A0          D0    A1          D1
```

```
BYTE OpenDevice(STRPTR, ULONG, struct IOStdIpcMem *, ULONG);
```

FUNCTION

This is an exec call. Exec will search for the name pointed to by `network_device` and if found will pass this call on to the device. For the sample implementation of the memory device the pointer `network_device` should point to "ipcmem.device".

INPUTS

`network_device` - pointer to literal string "ipcmem.device"
`unit` - Must be zero
`ioRequest` - pointer to an `ioRequest` block of size `IOStdReq` to be initialized by the `ipc.device`. (see `devices/ipc.h` for the definition)
`flags` - Must be zero for future compatibility

RESULTS

`D0` - same as `io_Error`
`io_Error` - If the Open succeeded, then `io_Error` will be null. If the Open failed, then `io_Error` will be non-zero.
`io_Device` - A pointer to whatever device will handle the calls for this unit. This pointer may be different depending on what unit is requested.

BUGS

SEE ALSO

ipcmem.device/CloseDevice

Outline of Producer/Consumer Programs

Using BSD Sockets:

```
Producer() {
    fd = socket(...); /* create basic connectionless */
                        /* datastructures */
    bind(fd, ... ); /* assign name for rendezvous */

    connection = accept(fd, ...); /* wait for consumer */

    write( connection, ... ); /* send some data */

    close( connection ); /* close connection */

    close( fd); /* remove rendezvous point */
}
Consumer() {
    connection = socket(...); /* create basic connectionless */
                        /* datastructures */
    connect(connection, ...); /* establish connection */
                        /* to waiting producer */
    read( connection, ... ); /* get the data */

    close ( connection );
}
```

Using Amiga Devices:

```
Producer() {
    OpenDevice ( protocol.device ); /* establish raw connect */

    DoCommand( IPCMD_AddObject, ... );

    /* present implementation of AddObject waits for */
    /* a connection to be attempted by consumer */
    /* assuming that for now. */

    DoCommand( IPCMD_Accept, ... ); /* verify connection */

    DoCommand( CMD_Write, ... ); /* send some data */

    DoCommand( CMD_CloseConnection, ... ); /* terminate */

    CloseDevice( protocol.device );
}
Consumer() {
    OpenDevice ( protocol.device ); /* establish raw connect */

    DoCommand( IPCMD_Connect, ... ); /* create connection */

    DoCommand( CMD_Read, ... ); /* get the data */

    DoCommand( CMD_CloseConnection, ... ); /* terminate */

    CloseDevice( protocol.device );
}
```

These examples are very skeletal. They do not contain any of the proper extra parameters that must be specified. They are only meant as examples of what network code will look like.



AmigaVision - The Amiga's Multimedia Construction Set

by John Campbell

This paper gives an overview of the purpose, functionality, and promise of AmigaVision.

Purpose of AmigaVision

The Amiga product line has clearly demonstrated success in certain professional video markets¹:

- ☐ 1st in Character Generation/Titling products purchased in the last 12 months (with a 23.64% share)
- ☐ 1st in Paint Systems purchased in the last 12 months (with a 13.86% share)
- ☐ 1st in 3D Modeling/Animation installed base and products purchased in the last 12 months (with shares of 49.03% and 67.35% respectively)
- ☐ 2nd in Paint Systems installed base (with a 14.59% share to Chryon's 15.96% share)
- ☐ 3rd in Character Generation/Titling installed base (behind Chryon and Quanta, with a 6.84% share)

The success is directly related to the Amiga's inherent advantages; video compatibility, graphics power, multitasking, and affordability. Analysts predict that by 1993 the desktop video market "will reach approximately \$4.8 billion, up from roughly \$907 million in 1989"².

Commodore views desktop video, or multimedia, as the big growth market of the 1990s. As the 1970s had mainframe data processing the 1980s had desktop publishing, the 1990s will unveil the power of multimedia. To be successful, you must sell solutions, not buzzwords. Commodore has chosen five specific areas where it has targeted multimedia solutions:

1. Business Presentations
2. Education
3. Training
4. Simulations
5. Specialty (Point of sale, public areas, etc....)

The multimedia approach has clear and significant advantages in each of these areas.

Business presentations. With multimedia, business presentations need not be stagnant and linear. They can be dynamic through the proper use of color, sound, animation and video. They can be interactive, branching along paths desired by the audience. These features tend to make presentations more memorable, enjoyable, and persuasive.

Education. Educational uses of multimedia are not limited to the values of "learn at your own pace" or as an educational perk. Multimedia offers the capabilities of video to allow the user to see, for instance, the Martin Luther King "I Have a Dream" speech, experience a blastoff at Cape Canaveral, or "conduct" a dangerous chemical experiment. Additionally, the new medium offers exciting opportunities for those who have difficulty learning from the standard methods of teaching. Possible students include the deaf, the adult illiterate, or others for whom the traditional educational systems have not been successful.

Training. Multimedia has distinct advantages over other forms of hands-on learning when used for employee training. Computer based training (CBT) offers a safer learning environment than training where expensive machinery or dangerous materials are involved. CBT reduces the need for training personnel, particularly in a situation where training will take place on multiple sites. Also, CBT makes it easy to evaluate a worker's skills before they have to rely on those skills.

Simulations. A number of companies and government agencies have a need for simulations apart from their actual experiments. They wish to show the desired events quickly and inexpensively. Video combined with animation can successfully fill these requirements.

Special Applications. Finally, public areas call out for a solution to convey information or sell product. Malls, museums, point of sale displays, even supermarkets have begun to use multimedia solutions to great advantage.

Software Solution

The Amiga's leadership in desktop video makes it an ideal multimedia tool for the five functional markets identified above. The only missing element is the software library to fill the multitude of requirements in the functional markets.

AmigaVision was designed as a tool to help fill this void. It is an iconic, multimedia authoring system for developing courseware. It uses an outline metaphor to organize, edit, and present information to the user. It takes advantage of the standards already in place in the Amiga marketplace and puts the immense power of the Amiga into the hands of the masses.

Functionality of AmigaVision

Design Decisions

AmigaVision has two primary design goals: to be intuitive and powerful. To make the product as intuitive as possible, we chose:

- ☐ Iconic vs. script language
- ☐ Outline vs. timeline format

These decisions make it easy to set up a slide presentation or edit a simple course, but not having an inherent script language could limit flexibility. We added the capability to run ARexx scripts or executable programs from within a course, effectively giving a door to greater functionality, if necessary. However, AmigaVision is a full programming language. AmigaVision includes control statements, math and string functions, and variable support, just like other languages. It differs from other programming languages not just in its interface, but also in that it directly supports audio-visual elements such as pictures, sound, speech, animation, and videodisc. AmigaVision also includes dBaseIII-compatible database capabilities which are useful for the record-keeping many multimedia applications require.

Structure of the Product

The product has six types of icons and four pull down menus. Courses are designed by placing icons in a tree-like structure, double clicking on these icons to select their attributes, and then viewing, editing, or saving the flow with the pull down menus. The icons represent the basic commands of the AmigaVision programming language.

The six icon types are: Control, Interrupt, Data, Wait, AV, and Module. The four pull down menus are: Project, Edit, Tools, and System.

The seven Control Icons allow you to control the flow of your program. They include: subroutine call, conditioned goto, goto, loop, exit loop, if then, and if-then-else. A common example would be to branch to a different part of your program given input from the user.

The three Interrupt Icons allow you to stop the flow of your program based upon user input. An example would be to give global help regarding how to use your program, or an ability to go back to the beginning of a program.

The seven Data Icons include three icons for dealing with the database, an icon for setting up variables, an output icon, and two icons for creating forms for user input.

The five Wait Icons are representations of waiting for a condition, mouse input, keyboard input, or some author defined time delay. AmigaVision also allows you to logically "and/or" these icons or time-out after an author-defined period of time.

The exciting part of AmigaVision is its nine AV Icons. In this section you are given control over pictures, animations, speech, digitized sounds, music, brushes, shapes, text files, and videodisc. By supporting the standards already set in the Amiga community, AmigaVision gives both novices and experts impressive control over the contents of their presentation. Transitions (such as fades and wipes) are supported to give extra flash to the product. To make control of the videodisc easier, a controller is included which is similar in depiction to a familiar hand held remote.

The last icon set is the Module group. These seven icons are more general in nature and aid the author in organizing the flow, managing resources, and in using software external to AmigaVision in a course. Included are module, subroutine, quit, return, execute, timer, and resource.

Two icons require some description: execute and resource. Execute opens the door to external programs via Workbench, CLI, or ARexx. Resource will cause data to be loaded ahead of time, so that, when the data is needed, it will be ready to use.

AmigaVision Menus

The icons are used in conjunction with the four pull down menus to create a course. The four menus are Project, Edit, Tools, and System.

The Project menu is where you create or load, save, print, or present your AmigaVision course. You also use this menu to create or install a run-time version of your course.

The Edit menu gives the author special capabilities to revise a course more easily: collect, copy, info, preview, telescope, and search. These features operate much like an outliner or word processor. For example, collect allows you to group a bunch of icons into one module. This can be helpful for very large flows. Telescope is a toggle switch which allows you to expand or contract any icon which has sub-icons or "children". Preview allows you to present the course from any selected icon, rather than the beginning of a flow.

The Tool menu allows you to directly access the Object Editor, Videodisc, and Database. The Object Editor allows you to put a variety of objects on the screen, including rectangle, polygon, line, circle, ellipses, text, brushes, variables, input pulls and text windows. The Videodisc Tool (also called the Videodisc Controller) allows the author to browse a videodisc and easily save a sequence to be used in a course.

The Database Tool allows you to either read in a file or create a new database file. The file format is dBaseIII-compatible. Through the use of this tool you can set up a database structure, choose key fields, and easily input data into those fields.

One tool that is not accessible from the menu is the Expression Editor. The Expression Editor allows you to create variables and expressions, which are useful for conditional evaluation. Thirty math functions and twelve logical operators are supported in the Expression Editor.

The System menu is for system wide changes and currently allows you to toggle Workbench on and off, provided the windows are closed and non-active.

The Promise

To the developer, the promise of AmigaVision is as varied and exciting as the number of software and hardware products on the Amiga. In addition to the value of an increased installed base which multimedia solutions will help create, there are specific areas of interest to developers:

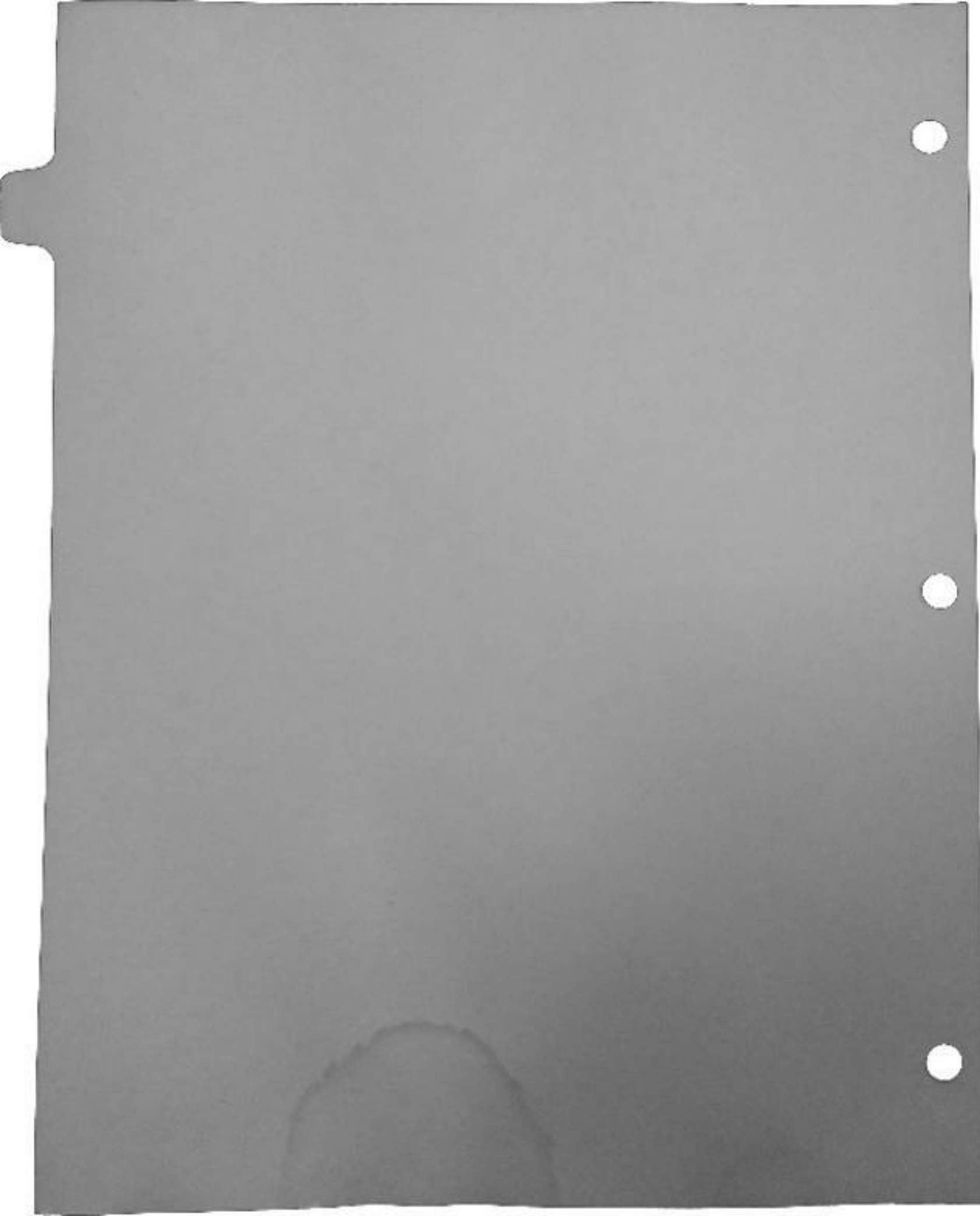
1. Creation Programs. AmigaVision is for presentation and manipulation of information which has already been created. It specifically *requires* other creation programs to have any value. Developers with strong creation products who support Amiga standards (as described in AmigaVision) will be listed in every AmigaVision manual which will be found with every CPU bundled with AmigaVision.
2. Courseware Developers can choose to use AmigaVision (possibly in conjunction with C or ARexx) to create powerful multimedia applications.
3. Add-on products. AmigaVision emphasizes the need for special add-on products such as videodisc players or touch screen drivers.
4. Prototypes or Training. With AmigaVision now being bundled, developers can make AmigaVision courses which demonstrate their products or help explain to purchasers how to use their products.

In conclusion, the Amiga is a machine whose enormous capabilities are as yet untapped. It is specifically well-positioned to dominate the growing multimedia market. Commodore has identified target markets and opportunities for developer involvement. With Commodore's new hardware and software offerings, we believe the Amiga and its developers can be successful in the years to come.

References

1. Sheer & Chaskelon Research Inc. 1989. "Amiga Market Shares in the Professional Video Market Place".
2. Presentation Business News, March/April 1990. "Study Forecasts Surge in Desktop Video by 1995".





Commodore Dynamic Total Vision: General Specifications

WARNING – *The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability.)*

DISCLAIMER – *This information is provided "as is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, incidental, special or consequential, resulting from any defect in the information, even if advised of the possibilities of such damages.*

System Overview

Commodore Dynamic Total Vision (CDTV) is an interactive consumer product cosmetically resembling a VCR, intended for the home entertainment environment. Easy to use via an infrared remote device (no keyboard required) and based on Amiga technology, it consists of a CD ROM drive (with CD Audio capability), a real time clock and remote controller. It can be easily connected to a TV or monitor and stereo system.

Target Market

In general, CDTV is targeted toward the family and the audio and entertainment enthusiast; and specifically toward educated adults, ages 24-49, who are interested in their own development and in providing their families with the greatest opportunity for advancement. This product is the multimedia device that allows them to use their TVs in a more intelligent and interactive manner by giving easy access to unlimited educational, informational and entertainment-oriented applications using the multimedia features of graphics and sound.

Major Functions

- ☐ Playing Amiga CD ROM software which can integrate various qualities of audio
- ☐ Playing CD quality audio while displaying graphics (CD+G)
- ☐ Playing standard high-fidelity CD-audio disks
- ☐ Can be expanded to an Amiga computer
- ☐ CD MIDI-in/MIDI-out

Technical Specifications

Central Processing Unit:

- ☐ MC68000 (Motorola) 16/32-bit main CPU
- ☐ 7.15909 MHz (NTSC)
- ☐ 7.09379 MHz (PAL) [NOTE: CPU speeds can vary when using an external system clock such as with genlocking units.]

Custom Chips:

- ☐ Three Amiga specific custom chips (Agnus, Paula, and Denise) which enhance system performance by taking over tasks such as handling video, sound, direct memory access (DMA), or graphics.
- ☐ CDTV will also have other custom chips for handling the CD ROM interface and infrared control.

Memory:

- ☐ 1MB chip RAM
- ☐ 2k non-volatile RAM reserved for system (clock, prefs, etc.)
- ☐ 512k ROM

Internal Slots:

- ☐ Intelligent video slot (for optional genlock, RF board, etc.); 15-pin edge connector
- ☐ DMA slot for SCSI

Video outputs:

- ☐ Analog RGB, Digital RGB (DB-23 connector)
- ☐ Composite video NTSC or PAL (RCA connector)
- ☐ Component video Y-C (S connector type for S-VHS and Hi8)
- ☐ RF Modulated (F connector)
- ☐ Optional genlock capabilities via plug-in module. Three-mode (CD, video source or mixed) under software control.

Video Display (General):

- ☐ 400 lines/vertical frequency 60Hz (NTSC)
- ☐ 512 lines/vertical frequency 50Hz (PAL)
- ☐ Graphic co-processor with beam synced draw, fill, and move modes (blitter)
- ☐ Maximum 1MB video memory (chip memory)
- ☐ Palette of 4096 colors
- ☐ Maximum 6 bitplanes
- ☐ 8 sprites per scanline

Text Modes:

- ☐ 80 characters/25 lines
- ☐ 60 characters/25 lines
- ☐ Various font sizes and types selectable
- ☐ User-definable screen colors

Graphics Modes:

NTSC

- ☐ 320 X 200 non-interlaced; 32 colors
- ☐ 640 X 200 non-interlaced; 16 colors
- ☐ 320 X 400 interlaced; 32 colors
- ☐ 640 X 400 interlaced; 16 colors
- ☐ plus overscan and HAM modes

PAL

- ☐ 320 X 256 non-interlaced; 32 colours
- ☐ 640 X 256 non-interlaced; 16 colours
- ☐ 320 X 512 interlaced; 32 colours
- ☐ 640 X 512 interlaced; 16 colours
- ☐ plus overscan, HAM, etc.

CD ROM Drive Specs:

- ☐ Sony/Philips type CD ROM standard mode 1, Mode 2
- ☐ Data readout from disk: 153k/sec (mode 1), 171k/sec (mode 2),
or 2MB/sec (burst)
- ☐ Average access time: 0.5 sec
- ☐ Maximum access time: 0.8 sec
- ☐ Soft read error: Less than 10e-9
- ☐ Hard read error: Less than 10e-12
- ☐ Seek error: Less than 10e-6
- ☐ Commands: CD ROM, CD-Audio, CD+G
- ☐ MTBF: 10,000 P.O.H.
- ☐ Standard supported: ISO-9660
- ☐ Data Capacity: 540MB (aprox) – [about 700 Amiga floppy disks]

CD Audio Specs:

- ☐ 8X oversampling
- ☐ Audio output: External 1.4VRMS, 10K OHM
- ☐ Frequency response: 20-20KHz
- ☐ Channel separation: -75DB (Typical)
- ☐ Harmonic distortion: 0.08% at 1 KHz
- ☐ Dual 16-bit D/A converter plus 64 levels of attenuation
- ☐ Maximum audio capacity: 14 hours - AM quality
- ☐ Sample Rates: variable from CD Audio rate (44Khz) to 6Khz

Computer Generated Audio Specs:

- ☐ 4 independent sound channels (right and left)
- ☐ Complex waveforms
- ☐ Sound buffer up to 128k, unlimited seamless sound
- ☐ 8-bit D/A converter plus 6-bit volume control

Audio Extras:

- ☐ Capable of sound mapping Amiga over CD-DA sound

Rear Ports:

- ☐ Centronics Parallel interface
- ☐ RS-232 Serial interface
- ☐ External floppy disk drive interface (Amiga floppy disk drive compatible)
- ☐ Hardwired alternative to IR for keyboard, mouse, joystick
- ☐ 2 audio output ports (RCA type plug) (requires external audio amplifier)
- ☐ MIDI-in/MIDI-out

Front Port:

- ☐ Stereo headphone jack
- ☐ Port for optional personal RAM Card (up to 64k)

Power Consumption:

- ☐ 50W (average AC100-240V, 50/60Hz)

Environmental Conditions:

- ☐ Operating temperature: 5° C to 40° C (non-condensing); 41° F to 104° F
- ☐ Operating humidity: 25% to 80% (non-condensing)

Front Panel Display:

- ☐ Fluorescent (white characters on black)
- ☐ Time, track, and volume level
- ☐ Clock remains lit when unit is turned off
- ☐ Display controllable by application

Front Panel Controls:

- ☐ Power On/Off
- ☐ Headphone volume Up/Down
- ☐ Play/Pause
- ☐ Stop
- ☐ Forward/Reverse - Scan/Skip
- ☐ CD/TV
- ☐ Reset

IR (Infrared) Remote Unit Specs:

- ☐ Proprietary high-speed IR signal
- ☐ Power supplied by 2 AA batteries
- ☐ 10 function keys plus Shift key (20 total)
- ☐ Up, Down, Left, Right movement button
- ☐ Two select keys
- ☐ CD audio – Reverse, Forward, Play/Pause, Headphone Volume and Stop keys
- ☐ Computer reset function

Optional Accessories:

- ☐ External floppy disk drive
- ☐ Trackball
- ☐ Joystick (third party)
- ☐ Keyboard
- ☐ MIDI in, out, through (third party)
- ☐ Personal RAM or ROM card
- ☐ Genlock
- ☐ Expansion module to house hard disk drive, modem, floppy disk drive
- ☐ Keyboard IR interface with trackball
- ☐ Keyboard
- ☐ Two-player IR interface

Added Raw Keyboard Codes:

- | | | |
|--|--|---|
| <input type="checkbox"/> 6C – Play/Pause | <input type="checkbox"/> 6E – Fast Forward | |
| <input type="checkbox"/> 6D – Stop | <input type="checkbox"/> 6F – Fast Reverse | ◆ |

1

2

3



Commodore Dynamic Total Vision: User Interface Design

This document is a working list of user-interface standards for CDTV (Commodore Dynamic Total Vision) and presents some of the reasoning behind these proposed standards. This list is by no means complete and will be expanded over the next few months.

The purpose of these standards is to promote a uniform look and feel to CDTV applications. What will this "uniform look and feel" involve? In the scenario presented here, many of the Amiga Intuition capabilities will be disallowed as being too complex or counter-intuitive to a novice home user. Recommendations will be presented for low-level issues such as a small number of fonts and the colors to be used. And high-level functions normally left to the application developer, such as control panels, list requesters and scrolling functions, will be supplied in a toolkit.

When developer choices of formats and modes of presentation are decreased, the user should see an increased uniformity across many different applications. Note that this is not intended to limit how the developer presents information or content. For instance, an animation or graphic selected by the user will be under complete control of the developer. Also note that the use of standard fonts and other elements of the UI standard should greatly reduce the level of effort for developers.

This document should be considered a work in progress. We invite your questions and comments. If you would like to become more involved in the creation of UI standards for CDTV, contact Gail Wellington at Commodore.

Characteristics

The User Interface for CDTV should reflect the target user population and environment; that is, it should be designed for use on TVs by unsophisticated and often brand-new computer users in a home recreational setting. Therefore, it should have the following characteristics:

- ☐ Ease of use.
- ☐ Uniform look and consistency of operation across all similar applications.
- ☐ Minimum documentation or intuitive operation in any situation.

TV-Specific Issues

Since CDTV is targeted for the home market and for use with a television, certain issues should be considered which relate to the estimated six- to eight-foot viewing distance and the limitations of TV resolution.

Foremost, remember that televisions are not the same as computer RGB monitors. Television is an interlaced, overscanned medium. What looks good on a monitor may look terrible on a TV set in the home. It is always a good idea to view your screens on a TV set before you commit them to compact disc. No matter how good your application is, if it doesn't look good on the home TV set then the users will be disappointed.

Fonts. Simple fonts such as Helvetica, Diamond and Times are recommended, but just about any font will work if it isn't too fancy in the first place. In completely unscientific tests, we found that anything smaller than 20-point type becomes difficult to read from a distance of more than ten feet. Also, broadcast television character generators almost always use anti-aliased fonts with a neutral-color outline. Ideally, the outline color is halfway between the character color and the background color. Outlines, borders and drop-shadows greatly improve readability. Pastel-colored fonts work better than bright or primary colors. Off-white works better than pure white. Yellows, greys, and pale blues seem to work best. *NOTE: We are commissioning a few sets of fonts in various point sizes. They will be designed specifically for CDTV and should be freely available to registered developers.*

Colors. Colors should be subdued rather than bright – bright colors tend to bleed on poorly adjusted TV sets. If you are using a paint program that has color values in a range from 0 to 15, we strongly recommend that no value go above 12 or 13. The only safe way to test colors and color combinations for display on an NTSC or PAL TV is by using a waveform monitor and keeping maximum values around 85% IRE. The best overall advice is to keep the color values below saturation and look at the results on a television set.

Nationality and user preferences selection. While some people may adjust preference settings – particularly nationality settings – most people will never touch them (if it ain't broke don't fix it). However, certain adjustments will have to be made by the user upon power-up or after a loss of power. We will try to make the transition between preferences selection and normal use as seamless as possible.

Television sizes. Bear in mind that some people will be using small TV sets. While we might have colored borders around the edges of the screen, developers should work within a "safe" area. We will be supplying the specific details on the dimensions of this "safe" area in the future – in the meantime, allow at least an inch on all borders (assuming you are using a standard Amiga monitor).

Television colors. Most people don't have their sets adjusted very well so don't rely on certain colors representing certain things. There is also the possibility that some people will be using black-and-white sets.

Input/Output Issues

Many of the keys toward keeping CDTV software consistent and easy-to-use are found in how input/output devices are handled. Listed below are some of the issues we've been exploring – most are just questions with no definitive answers as of yet. We're still open to ideas and suggestions.

Infrared device. As a positioning device, any remote unit will be awkward to use for fine adjustments. For that reason, we probably won't have gadgets for resizing, dragging, etc. As an alternative to forcing the user to precisely position a pointer, we felt it would be better to have the user cycle through lists of selectable options. We are still working on the specifics of these controls, including: establishing a set definition for key functions (such as "help" or "return to main screen"); how the IR device will be used to scroll through lists; and the control of text speed. Another consideration: there may come a time when an application requires some form of text entry and not everyone will own a keyboard.

Other input devices. How will people use input devices other than the IR remote (keyboard, trackballs, joysticks, etc.)? One of the first thoughts is that wherever possible these other devices should behave or emulate the same functions as those on the IR remote. For example, moving the trackball or joystick to the left or right should be the same as pressing the left or right arrow keys on the IR remote.

Output devices. One of the major challenges CDTV presents is the question of how the machine will interact with output devices.

- ☐ **Printing.** Since we are providing serial and parallel ports it is not unreasonable to assume that users will eventually want to print things. Where do they get the printer drivers? Will we supply drivers on the Welcome disc or will drivers be included on application discs? How will a user select a printer driver? Will we supply a screen dump utility that will work for any screen?
- ☐ **MIDI.** Since we are providing a MIDI-out port, how will users use it? Will it be entirely application-dependent or will there be occasions when a user will want to direct all audio output through a MIDI device?
- ☐ **Video.**
- ☐ **Personal memory cards.** This issue raises a number of big questions. What sort of DOS will the user be using? What sorts of functions will they need to perform (load, save, delete, copy, etc.)?

- ☐ Disk drives. The personal memory card questions also apply to disk drives (hard or soft). What sort of DOS do we supply? What sorts of commands will users need? What sort of interface should we supply?

Screen Issues

Here are some preliminary goals we have identified for the CDTV interface:

- ☐ No Guru screen allowed!
- ☐ Inform the user when disk I/O or processing is going on with something like a gears turning animation or a "Zzzzz," or "Working . . ." message. The goal is to use animations wherever possible.
- ☐ Support a standard way for the user to navigate menu trees.
- ☐ No pointing.
- ☐ No double-clicking.
- ☐ No dragging.
- ☐ No typing – all selections must be in a list or computable.
- ☐ No window/task switch – confine activity to a single window of activity.
- ☐ Where needed, a type of standard split screen could be used. A function could put the second window in one of the 1/4- or 1/9-screen-sized window quadrants. The border for the window would be fixed by the called function, and the user would not be able to move the new "window."
- ☐ Always provide an exit option

CDTV is heading away from the traditional Workbench-style interface. While Workbench is infinitely easier than an MS-DOS interface, it is still too complex and "computerish" for many living rooms. If we can accomplish all of the above, there will be:

- ☐ No manuals
- ☐ 3.2 Form
- ☐ A single screen (to user)
- ☐ No windows because the whole screen is the window (and thus no window borders and no window gadgets since there is no window to close or resize)
- ☐ No menus
- ☐ Ghosting (to indicate an option is not currently selectable)
- ☐ A uniform exit icon

Operation of the Machine

There are five possible screens, or scenarios, that the user may be presented with upon starting up CDTV. Which screen they first see will depend on certain conditions, such as cold boot, CD-audio disk in drive, CDTV disc in drive, etc.

Proposed CDTV Startup Screens

Conditions	Result
Cold boot Prefs not set in NVR (Non-Volatile RAM)	Set Preferences screen (Prefs)
Warm boot/reset Prefs set in NVR No disc present	Insert Disc prompt (ID)
Warm boot/reset Prefs set in NVR CD-Audio or CD+G disk present	Audio Control Panel (ACP)
Warm boot/reset Prefs set in NVR CDTV disc present	CDTV disc auto-boots
Warm boot/reset Prefs set in NVR unknown format or damaged disc present	Disc Error Message

The ID Screen. The ID screen would be similar to the "Insert WB" graphic that Amiga users are presented with on start-up. Behind the ID prompt we might have the color cycling disc graphic. Later, if there is a CD-Audio disc present (or CD+G with the graphics disabled), the user could return to this screen and see only the color cycling disc graphic without the "Insert Disc" prompt. Instead, the "Insert Disc" graphic would change to a loaded disc, ready to be activated.

With no disc present, the user has only one option from the ID screen: exit to the Prefs screen. Of course, they could always insert a disc as instructed. If there is a CD-Audio or CD+G disc present, but the user has exited the ACP back to the ID screen, they will have two slightly different options: exit to the Prefs screen, or select the "Activate Disc" icon (which, in this case, brings up the ACP screen). If the user inserts an unrecognized format or damaged disk they will be presented with the Disc Error Message.

There will be an exit icon on the main screen of the application which will bring the user back to the ID screen.

The Prefs Screen. When the user turns on the machine for the first time, or if there has been a power failure, then NVR will not have a copy of the user-specified preference settings. Anytime there is no copy of the user-specified prefs in NVR, the first screen the user sees

will be the Prefs screen. If there is a CDTV application disc present (or any disc for that matter), but prefs are *not* set in NVR, the user would be presented with the Prefs Screen *before* the application auto-boots. It is also possible for the user to access this screen at other times in case they wish to change settings for some reason (daylight savings time changes, for example).

The Prefs screen will always give the user four options: exit Prefs to the ID screen; adjust the screen centering; select nationality (key maps); or set the clock.

Once the Prefs settings are changed and the user exits Prefs, a copy of the user-selected preferences will be written to NVR. If NVR preferences are not set, and the user exits the Prefs screen without setting anything, the default preferences will be copied to NVR.

If there is a CDTV disc present when the user exits Prefs, he or she will return to the ID screen for a moment while the application auto-boots. If there is a CD-Audio or CD+G disc present when the user exits Prefs, he or she will return to the ID screen where the "activate disc" icon can be selected to get to the ACP.

The ACP Screen. The ACP is where the user will have options for playing back and controlling CD-Audio and CD+G discs. The specifics of the ACP will be supplied at a later date.

The Disc Error Message Screen. This provides a way to signal the user that there is a problem with a disk. Either there is a hard error on the disk or they have inserted an unreadable format disc. This message could be as simple as flashing the background color of the ID screen or as complex as a series of more specific error messages. Either way the system has to provide a way to signal the user that all is *not* well, rather than just locking up.

To the user, there are only two main screens, ID and Prefs. The ACP will not appear to the user unless there is a disc present. If there is a CD-Audio or CD+G disc present the ACP will appear to the user as if it were an auto-booting, standardized application screen, (even though it will be in ROM.) The two main screen options that the user can access without inserting a disc are the Prefs screen and the Insert Disc (ID) screen. Selecting "exit" from ID will bring up Prefs. Selecting "exit" from Prefs will bring up ID.

If there is a CDTV, CD-Audio, or CD+G disc present then the "activate disc" icon should be the pre-selected default icon.

The above startup scenario brings up some rules and questions for application developers:

- ☐ The system must be able to distinguish between a CDTV disc and a CD-Audio or CD+G disc.

- ☐ All applications must be auto-booting.
- ☐ One of the very first things that an application should check for is a match on key-map settings in Prefs. If the application does not provide a matching language version, then the application must provide the user with an alternate choice or set of choices.
- ☐ For the sake of consistency, CBM should provide developers with the code necessary to emulate the key-map selecting routine found in the Prefs screen. This code should be easily modifiable to ghost key-maps that the application does not currently support.
- ☐ Is there a need for a user to get to the ACP if there is no CD-Audio or CD+G disc present?
- ☐ What sort of Error message should we provide in ROM? Should there be more than one type of error message?

Welcome Disc

We are starting to put together ideas about what should be included on the Welcome Disc (the disc included with the machine). One item that should be included is a set of instructions on how to use the machine. It has also been suggested that we include application demos. These demos will perform two functions: demonstrate the capabilities of the machine and serve as advertising for developers.

Application-Specific Issues

Obviously, each developer will have their own ideas about how an application should perform, what it should do, and how to present it to the user. Some developers will completely take over the machine and do things their own way. Other developers will let the system and developer tools handle all aspects of presenting information to the users. In both cases we should still try and present the user with a consistent way of doing things.

Start up. As stated before, all applications should be auto-booting, and all applications should check the key-map settings at start-up.

Instructions. If possible, all applications should try to provide any necessary instructions on the disk itself. This can either be done at the start or through the Help function. It probably would be a good idea to provide a link to the Help function at all times.

Help Function. If possible (or necessary) applications should provide Help screens for the user. Since we will be designating one key on the IR remote as a Help function, users will probably try and ask for help no matter what the application. If the application does not require Help screens then either bring the instruction screens back up or de-activate the Help key function.

Error Messages. Somehow we should come up with a standard set of error messages to inform users when something is wrong (*Personal Memory Card Full, File Not Found, Memory Card Not Present, Printer Trouble, etc.*)

Key Maps. Ideally, all applications will be either language-independent or provide versions of the program in all languages. Unfortunately, this is probably not going to be the case. Developers should be aware that this machine will be sold in many countries. If an application does not currently support the language selected in preferences, the user must be presented with alternatives.

Exiting. This may be a minor issue in most cases because the machine will re-boot when a new disc is inserted – unless the program has specifically requested an override of the re-boot function for multi-disc applications. However, all applications should still try to exit cleanly. If any settings or preferences have been altered, then they should be reset before exiting.

New Ideas – and Sources for Ideas

This section presents various short lists of topics which are all associated with (a) the manner in which information is now currently presented on a computer screen, (b) the sources for ideas about new ways of presenting information, or (c) new ways which we might consider.

Sources for ideas. Look for ways in which information is currently presented in the real world. Following a real world metaphor makes the presentation more familiar to users. For instance, the desktop model is a real world metaphor.

Current Computer GUI Paradigms:

- ☐ Tabular data presentation;
- ☐ MFF scan of word indices or ordered “nodes”
- ☐ Allow user to sort on some keys – (fits the network “node” ideas)
- ☐ Landscape: movement through a landscape of icons, or pictorial representation of data
- ☐ Word Search/Indexed retrieval
- ☐ Browsing
- ☐ Menu, control panel
- ☐ Game type
- ☐ “Mass Comp” building blocks and connectors
- ☐ Logo – construction oriented
- ☐ Hypercard and hypertext

Sources of ideas, by product categories:

- ☐ Current Amiga products and ideas.
- ☐ Hypertext ideas
- ☐ MAC products, hypercard
- ☐ Other existing products: current GUI SQL front ends.
- ☐ Object-oriented graphics systems.
- ☐ CASE
- ☐ Educational stuff – Mac, Amiga

An examination of the physical world reveals a number of information sources that are easily tapped by individuals without computers:

- | | |
|---|---|
| <input type="checkbox"/> Phone book: white pages (by name); | <input type="checkbox"/> Atlas |
| yellow pages (by category and name) | <input type="checkbox"/> Landscape scenes – physical movement |
| <input type="checkbox"/> Encyclopedia | <input type="checkbox"/> TV preview guide |
| <input type="checkbox"/> Dictionary | <input type="checkbox"/> TV channels (switching) |
| <input type="checkbox"/> Catalog | <input type="checkbox"/> Microfilm |
| <input type="checkbox"/> Magazine | <input type="checkbox"/> Microfiche |
| <input type="checkbox"/> Library card catalog | <input type="checkbox"/> Television |
| <input type="checkbox"/> Rolodex cards | |

Starting Point for New GUI Paradigms and Metaphors

Possible paradigms:

- ☐ Bookshelf
- ☐ Carousel
- ☐ Map
- ☐ Store (shopping mall)
- ☐ Flipping channels

Possible metaphors:

- ☐ Table (flat fixed files)
- ☐ Fiche
- ☐ Rolodex
- ☐ Outline
- ☐ Network
- ☐ Map (2D)
- ☐ Landscape
- ☐ Hypercard
- ☐ Split Screen, of various types

DB Formats for data to be presented:

- | | |
|--|---|
| <input type="checkbox"/> Record | <input type="checkbox"/> Image data/pixel/bitmapped |
| <input type="checkbox"/> Network | <input type="checkbox"/> Object (e.g., CAD/CAM) |
| <input type="checkbox"/> ASCII text, with or without indexes | <input type="checkbox"/> Other graphics forms |

Presentation Functional Areas:

- ☐ Browse Paradigm: (eg, cockpit/aerial view/map/fiche window)
- ☐ Selection menu of fixed items: Icons/box selections/buttons/tools
- ☐ Network presentation: hypercard/hypertext; selection of next thing.
- ☐ Browse/select/order menu or tools, and mode selection.

Display Techniques. The physical information systems discussed in the previous section suggest a number of computerized display techniques. A number of existing database display techniques will also be itemized here.

- | | |
|--|---|
| <input type="checkbox"/> Table | <input type="checkbox"/> Network |
| <input type="checkbox"/> Forms | <input type="checkbox"/> Fiche |
| <input type="checkbox"/> Hypertext | <input type="checkbox"/> Map |
| <input type="checkbox"/> Hypercard | <input type="checkbox"/> Scene with hot spots |
| <input type="checkbox"/> Outline | <input type="checkbox"/> 3"x5" card with word search |
| <input type="checkbox"/> Rolodex | <input type="checkbox"/> Free flowing text with word search |
| <input type="checkbox"/> TV "channels" | <input type="checkbox"/> VCR |

Information System-Specific Issues

- ☐ Atomicity of data
- ☐ Network versus unstructured data: user presentation capability
- ☐ Presentation of selections from lists: visibility of indices and presentation modes for bypass of keyboard type selection.

Database Structures. The previous sections discussed possible presentation paradigms, which are the external features of a database. No discussion about databases would be complete without a list of possible internal database formats.

- ☐ Record (fixed or variable) – with or without indexes
- ☐ Network (e.g., Hypercard)
- ☐ Relational (interrelated flat files, with dictionary info)
- ☐ ASCII text (with or without indexes)
- ☐ Image (structured or bitmap)
- ☐ Object (e.g., CAD/CAM)
- ☐ Procedural (generates data dynamically)

Atomicity of Data. One of the broadest questions that must be answered before designing a new Visual Information System revolves around the "atomicity" of the data. This phrase that we have coined bears explaining. A database with a high degree of atomicity is highly regular and lends itself to many different display techniques. A database with a low degree of atomicity is more "hard wired;" its display paradigm is determined at construction time rather than at runtime.

For example, a database of automobile parts may be constructed with either a high or low degree of atomicity. One constructed with a high degree of atomicity might include the part number, the part name, the part type, the assembly name and the sub-assembly name for each and every part in the automobile. This database could then be displayed as a table, microfiche, outline, form or any other format available today or in the future.

A parts list constructed with a low degree of atomicity may be modeled on an existing parts catalog, with exploded views of each assembly displayed graphically. This "hard wired" format does not implicitly allow multiple display formats. This does not necessarily mean that a database with a low degree of atomicity is necessarily restricted to only one display paradigm, but is restricted to the paradigms allowed for at construction time. This is the way Hypercard works.

Some of us believe that providing powerful display paradigms which work with atomic data provides the most flexibility with the lowest cost for the data publishers, who construct the databases using the provided database tools. Others believe that providing extreme flexibility in the presentation of the data, and, in fact, the ability to present each and every piece of data in the database differently will provide the nicest, most application-specific presentations.

The final solution is probably to include aspects of both approaches. Solid and simple display paradigms should be available to the data publisher, while allowing the data publisher to tie together these displays with a more hard wired shell. In addition, it may be appropriate to provide a complete presentation language to the data publisher which would allow complete flexibility and an escape mechanism for future enhancements. ♦



Publishing and Selling CD-ROM Software

by Jim Mackonochie

There are now nearly 1000 products published and available on CD-ROM, and many more have been developed for use internally by large organizations for their own purposes. The products fall mainly into the following categories:

- ☐ Text databases
- ☐ Maintenance and service manuals
- ☐ Financial databases (i.e. LOTUS One Source)
- ☐ Clip art

The primary market for these products has been libraries and corporate institutions. Selling into these markets has been through direct sales or via the specialized CD-ROM distributors which have developed over the last few years (i.e. Bureau of Electronic Publishing, EBSCO, EDUCORP).

The primary source for CD-ROM material has been books or subsets of online files. Most CD-ROM products are strictly text with a limited target market. But there are a few multimedia products suitable for a mass market which have been developed and published on CD-ROM. Examples are:

- ☐ Defender of the Crown (IBM)
- ☐ Manhole (Macintosh)
- ☐ Cosmic Osmo (Macintosh)
- ☐ Whole Earth Catalog (Macintosh)
- ☐ Guinness Disc of Records (Macintosh & IBM)

These have also have been sold through specialized channels because there are not many consumers with CD-ROM drives, so mass merchandisers will not handle them. The notable exception is Egghead Discount Software, which now stocks a limited range of CD-ROMs. All this is about to change with the AMIGA/CD leading the entry of CD-ROM into the mass market.

The Characteristics of CD-ROM

CD-ROM is a storage device and hence a distribution medium - a high-storage-capacity distribution medium. The CD-ROM does not add anything more to the text, graphics, audio, and animation processing capabilities of the computer, however, the mass storage capacity gives opportunities to enhance existing products and create a whole new range of product types.

The economies of publishing on CD-ROM are attractive. Volume manufacturing costs of under \$1-a-disc are somewhat better than the costs of cartridges in the console market. The challenge is to create products to open up new sectors and new users in the leisure, education, and training markets.

Software Applications

The current entertainment markets can be categorized by user as follows:

Category 1 - Pre-teens & young teenagers

- ☐ Arcade games
- ☐ Sports simulations

Category 2 - Teenagers & adults

- | | |
|---|--|
| <input type="checkbox"/> Arcade games | <input type="checkbox"/> Vehicle simulations |
| <input type="checkbox"/> Sports simulations | <input type="checkbox"/> Environment simulations |
| <input type="checkbox"/> Adventures/RPG | <input type="checkbox"/> Board Game derivatives |

The bulk of the revenue generation is currently in category 1 in the USA, though this is not the case in Europe. The users worldwide are predominately male, though this characteristic is not so marked in the pre-teens.

The new generation of Amiga/CD products needs to emphasize different benefits for different users. For existing home computer and console users, the main benefit is that games and products that they already enjoy will be so much better and richer in experience with CD-ROM.

But Amiga/CD products should also be aimed at new users. To do this, new software products must be developed that are desirable to consumers who have never been attracted to the concept of using computer systems for recreation or self-improvement and learning. Most important is to create products that play to the strength of the CD technology, and create a market for the whole family.

CD-ROM Product Concepts

There are at least two areas that have just begun to be exploited by the computer industry and which also appeal to the family market: information- and audio-based products.

One obvious information product is an encyclopedia with powerful text retrieval software. Although 200,000 pages of plain text has its place, it is neither revolutionary nor exciting. Better information products will be a blend of text, pictures, audio, and animation. Those that explain, inform, demonstrate, and are fun will be the most desirable.

Remember that the Amiga/CD will be able to play all the mass market CD audio discs, one of the most successful mass market products that the consumer electronic market has seen. Use audio effectively in your products to create atmosphere, tension, and excitement. Learn the lessons from radio and TV and the range of emotions and images that audio can create.

To create a market for female teenagers, take a lesson from the success of pop videos. The CD-ROM technology cannot emulate a linear video, (for the moment) but I can think of a variety of products based on music and displaying graphics, animation, and text which are well within the current capabilities of CD-ROM.

CD-ROM Product Design

Although CD-ROM provides mass storage, there is a down side. The data transfer rate is limited to about 150K a second, and in the worst case it can take over a second to seek and find a file to load into RAM. One thing that will kill a product is for a user to be waiting for 5 seconds for something to happen when the expectation is that it should be instantaneous. These problems can be overcome with good product design and a knowledge of the computer system and medium.

Techniques to avoid seek time and transfer rate limitations include:

- ☐ Loading pointers into RAM on the start up of the application or during use of the application.
- ☐ Concatenating small files into a large file.
- ☐ Diverting the user while seeking and loading takes place.
- ☐ Preloading data which is likely to be used at the next stage of an application, while the user is manipulating the current stage.
- ☐ Making the best use of disc geography: putting data on the disc in close physical proximity to related data.
- ☐ Using of compression/decompression techniques to lessen the effect of a 150K transfer rate.

A key point in CD-ROM product design is not to force the user through unnecessary sections of a program. For example, online instructions can be especially useful to the novice user, but the experienced user may not need instructions. The user must have the option to bypass such a section.

Be aware that products are going to be sold in international markets. Currently, the sales of Amiga products are larger in Europe than in the USA. Should a product's screen be optimized for NTSC or PAL, or should both be included on the same disc?

Within Europe, the English speaking market is probably about 40%, but German and French speaking markets are significant. This is not a problem for arcade games, but will be for multimedia products with a significant text language and audio base. Use the CD-ROM capacity to create one multilingual product - it could claim it has educational advantages!

Future Trends for Distribution and Selling

With the launch into the mass market of multimedia machines such as the Amiga/CD, we can be confident that the software products on CD-ROM will become accepted and handled by the existing software distribution channels, it will be just another SKU.

However, it is possible that other non-computer distribution channels may also be attracted to handle CD-ROM products. These channels include books and audio/records, and possibly consumer electronic channels which have not previously handled "software".

Conclusion

While it is impossible to predict the near future with a high level of accuracy, one thing is sure. The whole industry is about to undergo a major change with the use of CD-ROM based systems. It offers the opportunity to create fantastic products, the likes of which we could only dream about 10 years ago when we were struggling with machines limited to 16K or less of RAM.

Hopefully it will be profitable; for sure, as always in our industry, it will be fun. ♦



V2.0 Preferences

by Eric Cotton
June 1990

The Commodore-Amiga Operating system has, through version 1.3, relied on a finite Preferences structure 232 bytes in length. This structure supports a limited definition of the user's working environment. The introduction of new peripherals, display modes, etc. as well as the attraction of supporting additional user preferences from both Commodore and third-party developers suggests the need to update Preferences in a flexible and extensible way. This article presents the new design of Preferences implemented in V2.0.

OVERVIEW

Preferences has undergone a complete facelift for V2.0. The most noticeable difference is that instead of one Preferences editor program there are now many. Additionally, there are many more user-environment settings over which the user has control.

Environment and Preferences

Preferences is now part of ENV:. In previous software releases we have indicated that ENV: will become an integral part of the Amiga OS. While developing the new Preferences it became apparent that Preferences and environment variables share many of the same goals and concepts. Thus the two have been merged together. By combining the two we hope to enhance the flexibility and usefulness of both. Use of the term "Preferences" continues, however, as the name applied to the customary system of programs and data which configure the Amiga's working environment. This includes the system editors, their data, IPrefs (the Intuition Preferences daemon), and any application programs choosing to use the features of Environment to organize and maintain configuration information.

As is the case today, ENV: will normally be assigned to a subdirectory of RAM:, specifically RAM:Env. This is where "in use" Preferences are stored. Most system Preferences can be found in the Sys directory in ENV:.

Preferences Data Files

Preferences items are usually grouped by type into distinct data files. These files reside in the logical device ENV:. Each file contains the Preferences data for a particular class of items. For example, one such file, serial.prefs contains the settings and configuration for the serial port.

Most Preferences data files are duplicated on the Workbench disk in the SYS:Prefs/Env-Archive directory, referenced with the assign ENVARC:. In the event of a system reset they are copied from their archival storage to ENV: whereupon notification is sent to interested parties such as Workbench.

Editors (Preferences Writers)

The editors are tools for changing Preferences items. When a user wishes to change settings he simply executes the appropriate editor located in the Prefs drawer on his WorkBench. While editors can vary in appearance, their internal operations are similar. When invoked, an editor opens and reads into its own private memory the data from the ENV: file on which it operates. This can be done by the familiar DOS functions. If the file does not exist, the editor should provide a default setting for each Preferences item (where appropriate). Once it has a complete set of data, it initializes its display to reflect the current settings. Then, a user interface similar to that of the V1.3 Preferences editor allows the user to modify the state of each item. The editor then saves the new data back to the data file in ENV: and, optionally, ENVARC:.

Customers (Preferences Readers) and Notification

A Preferences customer is any program that reads Preferences. Examples include terminal programs inquiring about the configuration of the serial port, word processors which need printer setup information, and even Workbench.

Previously, customers relied on Intuition and the NEWPREFS IDCMP to find out about Preferences changes. New programs, however, should utilize the new filesystem notification ability introduced in V2.0. Then each time an editor writes its data, the registered customer programs are notified that a change has been made.

Any process can become a customer and receive notification by calling the new DOS function StartNotify() specifying the name(s) of the data file(s) for which it would like to be informed. A message or signal is then sent back to the program when the file (or files) have been modified. The customer then has the option of opening the Preferences file to find any changes made to the data.

IPrefs

IPrefs is short for Intuition Preferences daemon. It is IPrefs job to communicate certain Preferences information from the system Preferences data files to Intuition.

COMPATIBILITY AND SYSTEM INITIALIZATION

A degree of compatibility with previous versions of Preferences and the OS must be maintained. In this regard, certain artifacts of the original Preferences and its interface have been preserved.

When the system boots off of a disk containing a DEVS:system-configuration file, the DOS will use the data in this file to initialize Intuition's internal Preferences data structure. All new V2.0 items will be reset to defaults. Thereafter, any appropriate Preferences files found in ENV: will override the items in the system-configuration file.

Intuition will continue to support the Preferences-related functions, GetPrefs(), GetDefPrefs(), and SetPrefs(). While applications are discouraged from using these functions they are nonetheless retained to insure compatibility with programs written before V2.0. They operate exactly as before but will only support the subset of Preferences data for for which Intuition keeps an internal record.

The SetPrefs() function (if Inform = TRUE) will still cause Intuition to send a message to those programs which have the NEWPREFS IDCMP flag set. Again, this mechanism only relates to changes in the Intuition Preferences buffer. GetPrefs() will report, to the best of its ability, only items which reflect the current state as set by SetPrefs(), and not necessarily by new Preferences or environment variables.

System Initialization

System initialization in V2.0 is as follows:

- ☐ DOS reads the DEVS:system-configuration file, if it exists, and calls SetPrefs() to set up Intuition's internal Preferences structure. New items are set to default values.
- ☐ The archived Preferences/Environment files (ENVARC:) are copied to ENV:

Copy >NIL: ENVARC: RAM:Env all quiet
- ☐ ENV: is assigned to the RAM:Env directory.
- ☐ The system configuration daemons are started up, and they register for notification of their favorite environment files. (Note that the Intuition daemon, IPrefs, is currently the only one.)

- ☐ The system daemons check for existing files, and if they exist open them. They pass along the contents to the interested system module.
- ☐ Intuition updates its internal configuration including (for compatibility) its Preferences structure.
- ☐ The Workbench screen opens for the first time.

PREFERENCES DATA FILES

The Preferences data files act as both storage for configuration information and as intermediaries between the editors and applications.

Data Format

There is no strict structure for Preferences files. They may be ASCII, IFF, binary, or any form suggested by the Preferences data therein. Many of the system files are comprised of structured binary data and are simply saved as such with a small amount of identifying header information. Some, such as `pointer.ilbm`, are actual IFF ILBMs.

Data in Use

Generally `RAM:`, and in particular `ENV:`, is where "in use" Preferences data is stored. `RAM:` is preferable over other storage media in that it offers both notification and extremely fast access to the data in the files.

The root of `ENV:` is reserved for environment variables. System preferences files, such as `printer.prefs`, are assigned to the `Sys` directory within `ENV:`. Applications which have their own Preferences are encouraged to put them in a directory named after their application. For instance, a paint program named `MyPaint` should put its Preferences files in a `ENV:MyPaint` directory. This will prevent a name collision if there are multiple applications which all call their palette preferences `colors.prefs`, for example.

Because `RAM` is a limited resource, applications which have very large amounts of Preferences data should not store it directly in `ENV:`. Instead, a "pointer" to the actual data (such as a filename) should be stored. Thus a screen backdrop picture should be saved on disk and the `ENV:` Preferences would contain little more than the name of the disk file and a directory path to it.

Archived Data Files

Clearly, Preferences saved to RAM: will not survive system reset. Consequently, all Preferences data should also be saved to non-volatile storage such as a hard disk. In this regard, a directory in the SYS:Prefs drawer has been created to archive copies of Preferences files. The directory is called Env-Archive; it may be referenced by the logical assign ENVARC:. ENVARC: is structured much like ENV: and in most respects is a copy. The content of the data files can differ (see the explanation of Use and Save in the Editors section).

The entire contents of ENVARC: is copied to ENV: by the startup sequence so applications need not be concerned about copying their Preferences data themselves. They should be careful, though, what they put in ENVARC: since it will find its way into valuable RAM.

Application programs may choose to put their Preferences files within the application's own directory. In that case, if notification is required then it is the application's responsibility to copy its Preferences data to an aptly named directory in ENV:.

Because there is a large overhead for the multitude of small files likely to be stored, an optional archiver (not included with V2.0) can be used to concatenate and perhaps compress the data. During a system boot, the files can then be extracted from the archive and returned to ENV:

Preference Presest

The directory Sys:Prefs/Presets is available for storage of Preset Preferences. Presets are explained in a following section.

PREFERENCES EDITORS

Until now the Preferences familiar to most Amiga users was the lone Preferences program which operated on a single data structure. Just to change one or two items often involved traversing multiple screens in the program. This has all changed in V2.0 with the advent of multiple System Preferences Editors. They are kept in the SYS:Prefs drawer on the Workbench disk.

System Editors

A family of editors are included with V2.0 to allow users to easily modify their working environment. While we anticipate that third-parties will eventually provide editors to manipulate their particular environment needs, the initial ones are the System Editors.

While most of the Preferences options available in V1.3 remain, the user now has much more control over his environment than ever before, including the ability to select multiple fonts, background patterns for the Workbench, overscan control, additional serial support, etc. But unlike the monolithic DEVS:system-configuration file of today, there are multiple files, each containing definitions for a particular facet of the system environment. Typically, editors exist in a one-to-one correspondence with each Preferences file.

Following is a list of the current system Preferences editors and their data files:

Font (screenfont.prefs, sysfont.prefs, wbfont.prefs) - Specifications of the default screen, system, and Workbench fonts (the latter is the font Workbench uses for icon labels).

IControl (icontrol.prefs) - Intuition specific control items including verify timeout, command key definitions, etc.

Input (input.prefs) - Mouse and keyboard control items.

Overscan (oscan.prefs) - Standard and text overscan areas for the various modes supported by this system.

Palette (palette.ilbm) - Color selections for the Workbench screen.

Pointer (pointer.ilbm) - Design of the mouse pointer image.

Printer (printer.prefs) - Printer text preferences as well as the printer driver name.

PrinterGfx (printergfx.prefs) - Printer graphic preferences.

ScreenMode (screenmode.prefs) - Display information such as the Workbench display mode and the raster dimensions.

Serial (serial.prefs) - Serial port definitions including baud rate, handshaking, parity, etc.

Time (none) - System and RTC clock date and time.

WbConfig (wbconfig.prefs) - Miscellaneous Workbench specific items.

WbPattern (win.pat, wb.pat) - The backdrop patterns used in the Workbench and its windows.

Access to files is usually accomplished via the normal DOS commands such as Open(), Read(), Write(), etc. Because many of the files are in IFF, the iffparse.library can be used to greatly simplify access to the data. Notification to customer programs is handled automatically when files are properly saved to ENV:.

Editor Design Guidelines

Design guidelines for system editors will insure that each presents a familiar and consistent user interface. Functionality is similar to the V1.3 Preferences program. New user interface features have been added, however, to more easily manipulate Preferences data. Editors are encouraged to include the following standard operations in menus and/or gadgets:

Project Menu

Open - Load data from a Preset file (see Presets, below). The user must specify a file's name as well as its path.

Save As - Save the data with a user-supplied name and path. The new file can be used as a "Preset" (see below). Customers will not necessarily be notified.

Quit - Exit the editor.

Edit Menu

Reset to defaults - Reset the editor display with a set of default data. Reasonable default values should be built in to each editor.

Last Saved - Reset the editor with the data in the appropriate .prefs file in ENVARC:. Note that the system editors retrieve the data from the ENVARC:Sys directory.

Restore - Reset the display to its original configuration (i.e. the original state when the editor was first invoked).

Undo (Optional) - Undo the most recent change the user has made.

Options Menu

Save Icons (yes/no) - Controls whether or not the editor will save a project icon with each preset saved (see Save As).

Editor Standard Gadgets

Save - Save the data to both ENV: and ENVARC: (archival storage) using the official name. Data is protected from system reset. Customers will be notified. The editor will terminate. Note that the system editors store their data in the ENV:Sys and ENVARC:Sys directories.

Use - Save the data to ENV: using the official name. Interested environment customers will be notified that a change has been made. Because no change is made to the copy in permanent storage, all changes will be lost if the system is reset. The editor will exit. Note that the system editors store their data in the ENV:Sys directory.

Cancel - Exit the editor, preferences are restored to the original state before the editor was invoked.

Editor CLI Usage

In addition to the user-friendly interface, many editors should also accept limited command line arguments when executed from the CLI. In this instance the editor will not always open its window. Instead, the editor will perform the action as detailed in the arguments and no further action will be taken. The command template should be of the form:

FROM,EDIT/S,USE/S,SAVE/S

FROM can specify a preset (the default FROM is the official file). The switches, EDIT, USE, and SAVE, specify the action the editor should perform as follows:

EDIT - Open the editor window and configure the display as directed by the data file (preset or otherwise). This is the default switch.

USE - Perform a "Use" on the data found in the data file. Do *not* open the editor window.

SAVE - Perform a "Save" on the data found in the data file. Do *not* open the editor window.

Here are two examples:

```
1> Serial FROM Sys:Prefs/Presets/myserial.pre USE
1> PrinterGfx Sys:Prefs/Presets/mypgfx.pre SAVE
```

In the first example the serial device Preferences editor is silently run from the CLI. Without opening its window the editor loads the myserial.pre serial preset file, performs a Use function and then exits. Likewise, in the second example the printer graphics editor runs silently but performs a Save function before exiting.

Preferences Presets

A new concept introduced in V2.0 is Preferences presets. Presets are alternate versions of the recognized Preferences files. Their purpose is to support varying configurations of Preferences which can be "turned on" at the user's discretion by selecting USE from its editor. For example, while the "accepted" definition of the mouse pointer is in pointer.ilbm, a user may have an alternate pointer he favors when working with a paint program. By saving the new image to a different name with Save As, he can at any point reload the image and select Use from within the Pointer editor and change pointers. As explained above, this can also be done from the CLI. The original version will still be available in the ENVARC:Sys directory. A summary of the procedure is as follows:

1. Execute the appropriate editor.
2. The current settings are automatically loaded (i.e. the file with the recognized name is loaded).
3. Change the settings to the new configuration.
4. Select Save As to store the new settings under a unique name.
5. Select Cancel. The new preset settings will not take effect. Instead, the original configuration remains in effect.

Then whenever the preset is needed simply:

6. Execute the editor.
7. Use the editor's Open menu item to retrieve the preset.
8. Select USE. (The preset is saved under the "recognized" name.)
9. The presets settings take effect.

Or from CLI:

6. Execute the editor with the USE switch while specifying the preset as the FROM file. The editor will perform a Use operation and terminate without opening its window.

Or from Workbench (assuming the preset was saved with an icon):

6. Double-click on the preset's icon. The editor (as identified in the preset icon's Default Tool) will perform a Use operation and terminate without opening its window.

As a convenience, the directory SYS:Prefs/Presets can be used to store presets. It is the default when selecting Save As from a system editor.

Utilities

It is recognized that various utilities may prove useful to facilitate some of the housekeeping associated with managing the various manifestations of the Preferences data, including, for compatibility, the DEVS:system-configuration file. We may in the future have a utility that will allow the user to create a system-configuration file, compatible with earlier versions of the OS, from the various system Preferences files.

PREFERENCES CUSTOMERS

A Preferences customer is any program that wishes to be informed of changes to Preferences items. Unlike the editors, which are the Preferences writers, the customers are the Preferences readers.

Customer programs which once relied on Intuition for notification should evolve to use the new filesystem notification system. A program can become a customer by asking for notification on any data file(s) stored in ENV: in which it is interested. Thereafter, any time a change is made to the data file, the handler will send notification to the customer program. The customer then has the option of opening the Preferences file to find any changes made to the data.

As an example, suppose a terminal program needs to know of any changes a user makes to the Preferences for the serial port. By simply registering the name of the serial Preferences file with the handler, the program will be notified every time the file is changed, including creation, removal, or name change.

Notification Options

Two types of notification are supported: notification by message and notification by signal. A customer that requests notification by message will be sent a NotifyMessage whenever the file changes. This message includes a pointer to the customer's NotifyRequest structure (see below) which will in turn have a pointer to the name of the Preferences file that has changed. Message notification is particularly useful when requesting notification on more than one file.

An additional feature of message notification is known as "throttling" (via the NRF_WAIT_REPLY flag). When throttling is enabled the handler will not send a second notification message for a given file until the application program has returned the first.

Besides preventing a buildup of messages it will also insure that the customer does not wastefully re-read the same data multiple times. The handler will, however, keep track of changes and, if necessary, immediately notify the program again after the message is returned.

A customer that requests notification by signal will just be signaled by the handler when the file changes and no message will be sent. While this method is faster than message notification, it is also much less informative. It is most useful when requesting notification on only one or two files. Throttling is inherently automatic.

Requesting and Removing Notification

To request notification a customer program should use the DOS function StartNotify() or its packet-level equivalent. A NotifyRequest structure must be passed to the handler. A pointer to this structure will be included in the NotifyMessage every time a specified Preferences file changes. Note that the structure will belong to the handler until notification is removed.

The synopsis for StartNotify() is:

```
BOOL StartNotify(struct NotifyRequest *)
```

The function returns the success or failure of the request. The NotifyRequest structure is listed below. See the C include file dos/notify.h and the DOS autodocs for more complete information.

```
/* Do not modify or reuse the notifyrequest while active. */
/* note: the first LONG of nr_Data has the length transferred */

struct NotifyRequest {
    UBYTE *nr_Name;
    UBYTE *nr_FullName;      /* set by dos - don't touch */
    ULONG nr_UserData;      /* for applications use */
    ULONG nr_Flags;

    union {
        struct {
            struct MsgPort *nr_Port; /* for SEND_MESSAGE */
        } nr_Msg;

        struct {
            struct Task *nr_Task;    /* for SEND_SIGNAL */
            UBYTE nr_SignalNum;      /* for SEND_SIGNAL */
            UBYTE nr_pad[3];
        } nr_Signal;
    } nr_stuff;

    nr_Reserved[4];          /* set to 0 */

    /* internal use by handlers */
    ULONG nr_MsgCount;
    struct MsgPort *nr_Handler; /* handler sent to (for EndNotify) */
};
```

The customer selects notification options by setting various flags in the `nr_Flags` field of the `NotifyRequest` structure. The current flags are:

```
#define NRF_SEND_MESSAGE    1 /* notification by message */
#define NRF_SEND_SIGNAL    2 /* notification by signal */
#define NRF_WAIT_REPLY     8 /* throttle notification messages */
#define NRF_NOTIFY_INITIAL 16 /* send initial notification */
```

The latter flag instructs the handler to immediately send notification (message or signal) on the selected files so that the customer may defer initial reading until after the program and notification have been set up.

To remove notification call `EndNotify()` (or its packet-level equivalent). The `NotifyRequest` structure will then be returned. The synopsis for this function is:

```
void EndNotify(struct NotifyRequest *)
```

Programs using message notification will receive a pointer to a `NotifyMessage`. Its structure is as follows:

```
struct NotifyMessage {
    struct Message nm_ExecMessage;
    ULONG nm_Class;
    UWORD nm_Code;
    struct NotifyRequest *nm_NReq; /* don't modify the request! */
    ULONG nm_DoNotTouch; /* like it says! */
    ULONG nm_DoNotTouch2;
};
```

Notification Example

The following code fragment is offered as an example customer requesting message notification of the file "foobar.prefs" in the ENV:myapp directory:

```
main()
{
    struct NotifyRequest *nr;
    struct NotifyMessage *nmsg;
    struct MsgPort *myport;
    ....
    nr->nr_Name = "ENV:myapp/foobar.prefs";
    nr->nr_UserData = 1234;
    nr->nr_stuff.nr_Msg.nr_Port = myport;
    nr->nr_Flags = NRF_SEND_MESSAGE|NRF_NOTIFY_INITIAL|NRF_WAIT_REPLY;
```

```

StartNotify(nr);
...
while (nmsg = (struct NotifyMessage *)GetMsg(myport))
{
    ...
    if (nmsg->nm_NReq->nr_UserData == 1234)
/*
* or you can use      if (nmsg->pm_NReq == nr)
* or                  if (strcmp(nmsg->pm_NReq->nr_Name, "foobar") == 0)
*/
    ...
    ...
    ReplyMsg(nmsg);
}
...
EndNotify(nr);
...
}

```

IPREFS

IPrefs is the Intuition Preferences daemon. Its purpose is to read some of the system Preferences files and pass on the information therein to Intuition. It was written to take advantage of the notification feature of RAM: much as an application program might.

IPrefs should be run from the startup sequence as follows:

```
Run >NIL: IPrefs >NIL:
```

It is most efficient to run the daemon after the Preferences files are copied from ENVARC: to ENV:. Because IPrefs sets the NRF_NOTIFY_INITIAL flag when requesting notification on the files, it will receive immediate notification.

Thereafter, each time a user selects 'Use' or 'Save' from within an editor, IPrefs will receive a notification message and, consequently, read and parse the data file. The information is then forwarded to Intuition. Where possible, Intuition will adjust itself to the new settings and also update its internal copy of the V1.3 Preferences structure. This will afford a degree of compatibility to those application programs which rely on GetPrefs() for their Preference information.

IPrefs currently reads and parses the following Preferences files:

- icontrol.prefs
- input.prefs
- overscan.prefs*
- palette.ilbm
- pointer.ilbm
- printer.prefs
- printergfx.prefs
- screenfont.prefs*
- screenmode.prefs*
- serial.prefs
- sysfont.prefs

Those marked with an asterisk require that The Workbench be reset in order for the Preferences to take effect. IPrefs will do this automatically, provided that all windows open on the Workbench screen are owned by Workbench. If there are other windows, IPrefs will, through a requester, ask that they be closed before it attempts to reset. Once IPrefs senses that the Workbench is clear of application windows it will attempt the reset. If, however, it fails to reset the Workbench in up to three successive tries then it will wait until the user explicitly selects 'Retry' (or 'Cancel') before attempting to do so again.

IPrefs may be terminated by sending it a CTRL-C. ♦



Using IFFParse in Applications

by Leo Schwab

IFFParse.library was created to help simplify the job of parsing IFF files. Unlike other IFF file libraries, IFFParse is not form-specific. This means that the job of interpreting the structure and contents of the IFF file is up to you. Previous IFF file parsers were either simple but not general, or general but not simple. IFFParse tries to be simple *and* general.

1. What's IFF *Really*?

Many people have a misconception that IFF means image files. This is not the case. IFF is a method of portably storing structured information in machine-readable form. The actual information can be anything, but the manner in which it is stored is very specifically detailed. This specification is the IFF standard.

It is recommended that you read the IFF standard. Then, read it again. Unfortunately, it is written in "standardese" and is difficult to understand in places. However, the standard does mention one powerful analogy: they compare an IFF file to a C program. If you keep this metaphor in mind, you will be well on your way to understanding IFF file structure.

One of the IFF FORM's that has been defined is the ILBM standard. This is how most Amiga bitmap imagery is stored. Since this is the most common IFF file, we'll be using this frequently as an example.

2. Handle Management

IFF files are manipulated through a grey box called an IFFHandle. The term grey box is used because only some of the fields in the IFFHandle are publicly documented. This handle is passed to all IFFParse functions, and contains the current parse state and position in the file. An IFFHandle is obtained by calling `AllocIFF()`, and freed through `FreeIFF()`. This is the only legal way to obtain and dispose of an IFFHandle.

3. Stream Management

A stream is a linear array of bytes that may be accessed sequentially or randomly. DOS files are streams. IFFParse uses 2.0 Hook structures (defined in utility/hooks.h) to implement a general stream management facility. Clients may implement any form of stream using this facility. IFFParse uses the facility to prepare a stream for use, read, write, seek, and terminate stream transactions.

On top of this facility, IFFParse has two built-in stream managers: One for unbuffered DOS files, and one for the Clipboard.

3.1. Initialization

In the IFFHandle is the public field `iff_Stream`. This is a longword that contains something meaningful to the stream manager. IFFParse never looks at this field itself (not directly, anyway). This field is initialized similar to the following:

```
iff->iff_Stream = (ULONG) OpenWeirdCustomStream ("foo");
```

In the case of the internal DOS stream handler, `iff_Stream` is a filehandle as returned by `Open()`:

```
iff->iff_Stream = (ULONG) Open ("filename", MODE_OLDFILE);
```

In the case of the internal Clipboard stream manager, `iff_Stream` is a pointer to a `ClipboardHandle`:

```
iff->iff_Stream = (ULONG) OpenClipboard (PRIMARY_CLIP);
```

(Note that `OpenClipboard()` is part of *IFFParse.library*.)

Once you've set up the `iff_Stream` field, you would then set the IFFHandle's flags and stream hook with `InitIFF()`:

```
InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &weirdstreamhook);
```

The flags are:

IFFF_FSEEK: This stream has forward-seek capability only.

IFFF_RSEEK: This stream has random-seek capability. (Note: this tends to imply `_FSEEK`, but it's best to specify both.)

If neither flag is specified, you're telling IFFParse that it can't seek through the stream.

There are calls to set up the internal DOS and Clipboard handlers:

```
InitIFFasDOS (iff); /* Sets up for DOS transaction. */
InitIFFasClip (iff); /* Sets up for Clipboard transaction. */
```

IFFParse "knows" the seek capabilities of these streams, so it sets the flags for you. *Note:* IFFParse sets IFFF_FSEEK | IFFF_RSEEK for DOS files. This is not always true (e.g. pipes). If the application happens to know that a DOS file has different seek characteristics, you may manipulate the seek bits in iff_Flags yourself after calling InitIFFasDOS().

Once initialized, call OpenIFF():

```
error = OpenIFF (iff, IFFF_READ);
```

Once you establish a read/write mode (by passing IFFF_READ or IFFF_WRITE), you remain in that mode until you call CloseIFF().

3.2. Termination

Termination is simple, just call CloseIFF(). This may be called at any time, and terminates IFFParse's transaction with the stream. The stream itself is not closed. The IFFHandle may be reused; you may safely call OpenIFF() on it again. You are responsible for closing the streams you opened.

3.3. Internals

If you are implementing your own custom stream handler, you will need to know the mechanics of hook call-backs, and how to interpret the parameters. The example program LookI contains an example of a custom stream call-back. You may use it as a reference (it's well-commented).

4. Parsing

This is both simple and complicated. It's simple in that it's just one call. It's complicated in that you have to seize control of the parser to get your data.

The parser operates automatically, scanning the file, verifying syntax and layout rules. If left to its default behavior, it will scan through the entire file until it reaches the end, whereupon it will tell you that it got to the end.

The whole show is controlled through one call:

```
error = ParseIFF (iff, controlmode);
```

The control modes are `IFFPARSE_SCAN`, `IFFPARSE_STEP`, and `IFFPARSE_RAWSTEP`. For the time being, we will consider only `IFFPARSE_SCAN`.

4.1. Controlling Parsing

`ParseIFF()`, if left to itself, wouldn't do anything useful. Ideally, we want it to stop at strategic places so we can look at the scenery. Here's where it can get complicated.

There are many tools provided to help control the parsing process. They range from simple and mindless to extremely deep and thought-provoking. You'll probably seldom use the deep and thought-provoking ones, so we'll cover the common ones.

4.1.1 StopChunk()

You can instruct the parser to stop when it encounters a specific IFF chunk by using the function `StopChunk()`:

```
error = StopChunk (iff, ID_ILBM, ID_BODY);
```

When the parser encounters the requested chunk, parsing will stop, and `ParseIFF()` will return the value zero. The stream will be positioned ready to read the first data byte in the chunk. You may then call `ReadChunkBytes()` or `ReadChunkRecords()` to pull the data out of the chunk.

You may call `StopChunk()` any number of times for any number of different chunk types. If you wish to identify the chunk on which you've stopped, you may call `CurrentChunk()` to get a pointer to the current `ContextNode`, and examine the `cn_Type` and `cn_ID` fields.

Using `StopChunk()`, you can parse IFF files in a manner very similar to the way you're probably doing it now, using a state machine. However, this would be a terrible underuse of `IFFParse`.

4.1.2. PropChunk()/FindProp()

In the case of a FORM ILBM, certain chunks are defined as being able to appear in any order. Among these are the BMHD, CMAP, and CAMG. Typically, BMHD appears first, followed by CMAP and CAMG, but you can't make this assumption. The IFF *and* ILBM standards require you to assume these chunks will appear in any order. So ideally, what you'd like to do is collect them as they arrive, but not do anything with them until you actually need them.

This is where PropChunk() comes in. The syntax for PropChunk() is identical to StopChunk():

```
error = PropChunk (iff, ID_ILBM, ID_BMHD);
```

When you call ParseIFF(), the parser will look for chunks declared with PropChunk(). When it sees them, the parser will internally copy the contents of the chunk into memory for you before continuing its parsing.

When you're ready to examine the contents of the chunk, you use the function FindProp():

```
StoredProperty = FindProp (iff, ID_ILBM, ID_BMHD);
```

FindProp() returns a pointer to a StoredProperty, which contains the chunk size and data. If the chunk was never encountered, NULL is returned.

This permits you to process the property chunks in any order you wish, regardless of how they appeared in the file. This provides much better control of data interpretation and also reduces headaches.

4.2. Putting It Together

With just StopChunk(), PropChunk(), and ParseIFF(), you can write a viable ILBM display program. Since IFFParse knows all about IFF structure and scoping, such a display program would have the added ability to parse complex FORMs, LISTs, and CATs and attempt to find imagery buried within.

Such an ILBM reader might appear as follows:

```
iff = AllocIFF ();
iff->iff_Stream = Open ("shuttle dog", MODE_OLDFILE);
InitIFFasDOS (iff);
OpenIFF (iff, IFFF_READ);
PropChunk (iff, ID_ILBM, ID_BMHD);
PropChunk (iff, ID_ILBM, ID_CMAP);
PropChunk (iff, ID_ILBM, ID_CAMG);
StopChunk (iff, ID_ILBM, ID_BODY);
ParseIFF (iff, IFFPARSE_SCAN);
if (bmhdprop = FindProp (iff, ID_ILBM, ID_BMHD))
    configurescreen (bmhdprop);
else
    die ("No BMHD, no picture.");
if (cmapprop = FindProp (iff, ID_ILBM, ID_CMAP))
    setcolors (cmapprop);
else
    usedefaultcolors ();
if (camgprop = FindProp (iff, ID_ILBM, ID_CAMG))
    setdisplaymodes (camgprop);
decodebody (iff);
showpicture ();
CloseIFF (iff);
Close (iff->iff_Stream);
FreeIFF (iff);
```

Note that error checking, of which there should be copious amounts, is not present above for clarity.

4.3. Other Features

There are other tools available for controlling the parser. Briefly, these are:

CollectionChunk(): PropChunk() keeps only one copy of the declared chunk (the one currently in scope). CollectionChunk() collects and keeps *all* copies of the declared chunk it encounters. This is useful for chunks such as CRNG.

StopOnExit(): Whereas StopChunk() will stop the parser just as it enters the declared chunk, StopOnExit() will stop just before it leaves the chunk. This is useful for finding the end of FORMs, which would indicate that you've collected all possible data in this FORM and may now act on it.

EntryHandler(): This is used to install your own custom chunk entry handler. StopChunk(), PropChunk(), and CollectionChunk() are internally built on top of this.

ExitHandler(): This is used to install your own custom chunk exit handler. StopOnExit() is internally built on top of this.

5. Reading Chunk Data

To read data from a chunk, use the functions `ReadChunkBytes()` and `ReadChunkRecords()`. Both calls truncate attempts to read past the end of a chunk. For odd-length chunks, the parser will skip over the pad bytes for you.

6. Writing IFF Files

IFFParse provides facilities for writing IFF files. Again, IFFParse makes no assumptions about the data you're writing, and concerns itself with verifying the syntax of your output.

6.1. Creating Chunks In A File

Because the IFF specification has nesting and scoping rules, you can nest chunks inside one another. One instance is the BMHD chunk, which is commonly nested inside a FORM chunk. Thus, it is necessary for you to inform IFFParse when you are starting and ending chunks.

6.1.1. PushChunk()

To tell IFFParse you are about to begin writing a new chunk, you use the function `PushChunk()`:

```
error = PushChunk (iff, ID_ILBM, ID_BMHD, chunksize);
```

The chunk ID and size are written to the stream. IFFParse will enforce the chunk size you specified; attempts to write past the end of the chunk will be truncated. If, as a chunk size argument, you pass `IFFSIZE_UNKNOWN`, the chunk will be expanded in size as you write data to it.

6.1.2. PopChunk()

When you are through writing data to a chunk, you complete the write by calling `PopChunk()`:

```
error = PopChunk (iff);
```

If you wrote fewer bytes than you declared with `PushChunk()`, `PopChunk()` will return an error. If you specified `IFFSIZE_UNKNOWN`, `PopChunk()` will seek backward on the stream and write the final size. If you specified a chunk size that was odd, `PopChunk()` will write the pad byte automatically.

`PushChunk()` and `PopChunk()` nest; every call to `PushChunk()` must have a corresponding call to `PopChunk()`.

6.2. Writing Chunk Data

This is done with `WriteChunkBytes()` and `WriteChunkRecords()`. If you specified a valid chunk size when you called `PushChunk()`, `WriteChunkBytes()` and `WriteChunkRecords()` will truncate attempts to write past the end of the chunk.

6.3. Example

Code to write an ILBM file might take the following form:

```
iff = AllocIFF ();
iff->iff_Stream = Open ("foo", MODE_NEWFILE);
InitIFFasDOS (iff);
OpenIFF (iff, IFFF_WRITE);
PushChunk (iff, ID_ILBM, ID_FORM, IFFSIZE_UNKNOWN);
PushChunk (iff, ID_ILBM, ID_BMHD, sizeof (struct BitMapHeader));
WriteChunkBytes (iff, &bmhd, sizeof (bmhd));
PopChunk (iff);
PushChunk (iff, ID_ILBM, ID_CMAP, cmapsize);
WriteChunkBytes (iff, cmapdata, cmapsize);
PopChunk (iff);
PushChunk (iff, ID_ILBM, ID_BODY, IFFSIZE_UNKNOWN);
packwritebody (iff);
PopChunk (iff);
PopChunk (iff);
CloseIFF (iff);
Close (iff->iff_Stream);
FreeIFF (iff);
```

Again, error checking is not present for clarity.

7. A Note On Seekability

As you can see from the above examples, IFFParse works best with a stream that can seek randomly. However, it is not possible to seek on some streams (e.g. pipes).

IFFParse will read and write streams with limited or no seek capability. In the case of reading, only forward-seek capability is desirable. Failing this, IFFParse will fake forward seeks with a number of short reads.

In the case of writing, if the stream lacks random seek capability, IFFParse will buffer *the entire contents* of the file until you do the final `PopChunk()`, or when you `CloseIFF()` the handle. At that time, the entire stream will be written in one go. This buffering happens whether or not you specify all the chunk sizes to `PushChunk()`. (Note: The current implementation of this internal buffering is very inefficient). Be aware that we reserve the right to alter this behavior of the parser, to improve performance or reduce memory requirements. We mention this behavior on the off chance it is important to you.

8. Context Utilities

Internally, IFFParse maintains IFF nesting and scoping context via a "stack." (Note that it is probably easier to think of a stack of things on a table in front of you when reading this discussion). It is from this stack concept that we derived the nomenclature `PushChunk()` and `PopChunk()`. Direct access to this stack is not allowed. However, many tools are provided to assist in examining and manipulating the context stack.

As the nesting level increases (as would happen when parsing a nested LIST or FORM), the depth of the context stack increases; new elements are added to the top. When these contexts expire, the `ContextNodes` are deleted and the stack shrinks.

8.1. `CurrentChunk()`

The current context is said to be the top element on the stack. Contextual information is stored in a grey-box structure called a `ContextNode`. You can obtain a pointer to the current `ContextNode` through the function `CurrentChunk()`:

```
currentnode = CurrentChunk (iff);
```

The `ContextNode` tells you the type, ID, and size of the currently active chunk. If there is no currently active context, NULL is returned.

8.2. `ParentChunk()`

To find the parent of a context, you call `ParentChunk()` on the relevant `ContextNode`:

```
parentnode = ParentChunk (currentnode);
```

If there is no parent context, NULL is returned.

8.3. The Default Context

When you first obtain an `IFFHandle` through `AllocIFF()`, a hidden default context node is created. You cannot get direct access to this node through `CurrentChunk()` or `ParentChunk()`. However, using `StoreLocalItem()`, you can store information in this context.

9. Context-Specific Data

ContextNodes can contain user data specific to that context. These data objects are called LocalContextItems. LocalContextItems (LCIs) are a grey-box structure which contain a type, ID, and identification field. LCIs are used to store context-sensitive data. This data can be anything. A ContextNode can contain as many LCIs as you want. LCIs normally won't be dealt with directly (use FindProp() and FindCollection() instead).

9.1. A Very Brief Overview of LCIs

The IFFHandle contains a list of ContextNodes. Each ContextNode contains a list of LocalContextItems. Anything that hangs off an LCI is user-defined.

When you call FindProp(), you are actually calling a front-end to FindLocalItem(). FindLocalItem() starts at the current context and searches all LCIs in that context. If none matching the specified type, ID, and ident are found, it proceeds down the context stack to the next ContextNode and searches all its LCIs. The process repeats until it finds the desired LCI (whereupon it returns a pointer to it), or reaches the end without finding anything (where it returns NULL).

Note that LCIs higher in the stack will "override" lower LCIs with the same type, ID, and ident. This is how we handle property scoping. As ContextNodes are popped off the context stack, all its LCIs are deleted as well.

StoredProperties (as returned by FindProp()) and chunk handlers are implemented internally as LCIs, which means they too obey the same nesting and scoping rules.

10. Error Handling

IFFParse's handling of errors is less than ideal. If at any time during reading or writing you encounter an error, the IFFHandle is left in an undefined state. Upon detection of an error, you should perform an abort sequence and CloseIFF() the IFFHandle. Once CloseIFF() has been called, the IFFHandle is restored to sanity and may be reused.

Error recovery is being investigated for a future version of the parser.

11. Conclusion

Between this document and the examples, you should have enough knowledge to read and write any valid IFF file out there. The FORM type we used most often for testing was FORM ILBM, because it's the most common. However, we also kept an eye towards other IFF file types that currently exist or may exist in the future.

A minor warning: IFFParse is still evolving in subtle ways. It is hoped that these changes will be complete by the time you're ready to use it. None of the "common" stuff will change. However, you may wish to watch out for changes in the implementation of EntryHandler(), ExitHandler(), and SetLocalItemPurge(); we're thinking of re-doing a call-back detail. However, *please* do not let this discourage you from using the parser; unless you're doing something *extremely* esoteric, these changes will *not* affect you.

Working with IFF has become much easier now that the messy job of parsing has been taken out of the programmers hands. Attention can be focused on higher-level issues which have heretofore been ignored or addressed poorly. Things like the X and Y aspect fields in the BMHD, what CAMG is *really* for, how to write a really neat BODY decoder, etc. will get the attention they deserve.

It is our hope that *IFFParse.library* will enable the IFF file specification to realize its full potential, and encourage Amiga programmers to take advantage of that potential and utilize IFF more frequently and more responsibly in the future.

Stuart H. Ferguson
123 James Ave.
Redwood City, CA 94062
USENET:
 well!shf@ucbvax.Berkeley.EDU

Leo L. Schwab
23 Summerhill Court
Terra Linda, CA 94903-3873
USENET: well!ewhac@ucbvax.Berkeley.EDU
BIX: ewhac
PLink: ewhac
Phone: (415) 472-1795



Dpaint ANIM Brush IFF Format

by Dan Silva

The ANIM Brushes of DPaint III are saved on disk in the IFF ANIM format. Basically, an ANIM Form consists of an initial ILBM which is the first frame of the animation, and any number of subsequent "ILBMs" (which aren't really ILBMs) each of which contains an ANHD animation header chunk and a DLTA chunk comprised of the encoded difference between a frame and a previous one.

To use ANIM terminology (for a description of the ANIM format, see the IFF ANIM Spec, by Gary Bonham), ANIM Brushes use a type 5 encoding, which is a vertical, byte-oriented delta encoding (based on Jim Kent's RIFF). The deltas have an interleave of 1, meaning deltas are computed between adjacent frames, rather than between frames 2 apart, which is the usual ANIM custom for the purpose of fast hardware page-flipping. Also, the deltas use Exclusive OR to allow reversible play.

However, to my knowledge, all the existing ANIM players in the Amiga world will only play type 5 ANIMs which have an interleave of 0 (i.e. 2) and which use a Store operation rather than Exclusive OR, so no existing programs will read ANIM Brushes. The job of modifying existing ANIM readers to read ANIM Brushes should be simplified, however.

Here is an outline of the IFF Form structures output by DPaint III as an ANIM Brush. The IFF Reader should of course be flexible enough to tolerate variation in what chunks actually appear in the initial ILBM.


```

FORM ANIM
  FORM ILBM                                first frame
    BMHD
    CMAP
    DPPS
    GRAB
    CRNG
    CRNG
    CRNG
    CRNG
    CRNG
    CRNG
    DPAN                                my own little chunk
    CAMG
    BODY
  FORM ILBM                                frame 2
    ANHD                                animation header chunk
    DLTA                                delta mode data
  FORM ILBM                                frame 3
    ANHD                                animation header chunk
    DLTA                                delta mode data
  FORM ILBM                                frame 4
    ANHD                                animation header chunk
    DLTA                                delta mode data
  ...
  FORM ILBM                                frame N
    ANH                                animation header chunk
    DLTA                                delta mode data

```

Here is the format of the DPAN chunk:

```

typedef struct {
  UWORD version;    /* current version=4 */
  UWORD nframes;    /* number of frames in the animation.*/
  ULONG flags;      /* Not used */
} DPANIMChunk;

```

The version number was necessary during development. At present all I look at is "nframes".

Here is the ANHD chunk format:

```
typedef struct {
    UBYTE operation; /* =0 set directly
        =1 XOR ILBM mode,
        =2 Long Delta mode,
        =3 Short Delta mode
        =4 Generalize short/long Delta mode,
        =5 Byte Vertical Delta (riff)
        =74 (Eric Grahams compression mode)
    */

    UBYTE mask; /* XOR ILBM only: plane mask where data is*/
    UWORD w,h;
    WORD x,y;
    ULONG abstime;
    ULONG reltime;
    UBYTE interleave; /* 0 defaults to 2 */
    UBYTE pad0; /* not used */
    ULONG bits; /* meaning of bits:

    bit#    =0                =1

    0  short data            long data
    1  stor                  XOR
    2  separate info        one info for
        for each plane      for all planes
    3  not RLC              RLC (run length encoded)
    4  horizontal           vertical
    5  short info offsets   long info offsets

    -----*/
    UBYTE pad[16];
} ANIMHdr;
```

For ANIM Brushes, I set:

```
animHdr.operation = 5; /* RIFF encoding */
animHdr.interleave = 1;
animHdr.w          = curANIMBr.bmob.pict.box.w;
animHdr.h          = curANIMBr.bmob.pict.box.h;
animHdr.reltime    = 1;
animHdr.abstime    = 0;
animHdr.bits       = 4; /* indicating XOR */
```

-- everything else is set to 0.

Note: the bits field was actually intended (by the original creator of the ANIM format, Gary Bonham of SPARTA, Inc.) for use only with compression method 4. I am using bit 2 of the bits field to indicate the Exclusive OR operation in the context of method 5, which seems like a reasonable generalization.

For an ANIM Brush with 10 frames, there will be an initial frame followed by 10 Delta's (i.e. ILBMS containing ANHD and DLTA chunks). Applying the first Delta to the initial frame generates the second frame, applying the second Delta to the second frame generates the third frame, etc. Applying the last Delta thus brings back the first frame.

The DLTA chunk begins with 16 LONG plane offsets, of which DPaint only uses the first 6 (at most). These plane offsets are either the offset (in bytes) from the beginning of the DLTA chunk to the data for the corresponding plane, or zero, if there was no change in that plane. Thus the first plane offset is either 0 or 64.

(The following description of the method is based on Gary Bonham's rewording of Jim Kent's RIFF documentation.)

Compression/decompression is performed on a plane-by-plane basis.

Each byte-column of the bitplane is compressed separately. A 320x200 bitplane would have 40 columns of 200 bytes each. In general, the bitplanes are always an even number of bytes wide, so for instance a 17x20 bitplane would have 4 columns of 20 bytes each.

Each column starts with an op-count followed by a number of ops. If the op-count is zero, that's ok, it just means there's no change in this column from the last frame. The ops are of three kinds, and followed by a varying amount of data depending on which kind:

1. SKIP - this is a byte with the hi bit clear that says how many rows to move the "dest" pointer forward (i.e. to skip). It is non-zero.
2. DUMP - this is a byte with the hi bit set. The hi bit is masked off and the remainder is a count of the number of bytes of data to XOR directly. It is followed by the bytes to copy.
3. RUN - this is a 0 byte followed by a count byte, followed by a byte value to repeat count times, XOR'ing it into the destination.

Bear in mind that the data is compressed vertically rather than horizontally, so to get to the next byte in the destination you add the number of bytes per row instead of one.

The Format of DLTA chunks is as described in section 2.2.2 of the ANIM Spec. The encoding for type 5 is described in section 2.2.3 of the ANIM Spec.



ECS Display Modes and ILBM CAMG

by Carolyn Scheppner

Under previous versions of the Amiga operating system and hardware, the available display modes such as HIRES, LACE, HAM, HALFBRITE, DUALPF, and assorted combinations of these display modes could all be described in a 16-bit ViewPort mode field (often referred to as a viewmode).

The 1.3 procedure for storing an Amiga viewmode in an ILBM was to take the 16-bit viewmode, mask out the undesirable bits GENLOCK_AUDIO, GENLOCK_VIDEO, VP_HIDE, and SPRITES, and store the result as a long (32-bit) value in a CAMG chunk (the upper 16 bits of the long would be 0).

The 1.3 procedure for reading an ILBM CAMG chunk was to read the 32-bit value, mask out the undesirable bits mentioned above, and then use the low word of the result as a 16-bit value for ViewPort.Modes or NewScreen.Modes when opening a display.

The 2.0 operating system and the ECS chips support an extensible number of display modes - too many to be specified by the previous method where individual bits mapped to a limited number of specific display characteristics. Under 2.0, display modes are now specified by a 32-bit ModeID which is not stored in the ViewPort, but is instead held in private graphics lists of extended display information linked into an extension of the ColorMap structure. System functions are provided for accessing this extended display information. A simple 2.0 function is provided for getting the new 32-bit ModeID for any ViewPort:

```
ULONG modeID = GetVPMoDeID(struct ViewPort *vp);
```

The 2.0 scheme for CAMG is to save the entire 32-bit ModeID, untouched, in the CAMG chunk.

Although ModeIDs are numeric values rather than bit masks, the current 2.0 ModeIDs have been specially designed to contain compatible old-style bits in their low word for the matching (or closest match) old ViewPort mode.

For example, the 2.0 ModeID for a Hires-Interlace display has the HIRES and LACE bits set in its low word. And the 2.0 ModeID for Productivity-Interlace also has the HIRES and LACE bits set in its low word. Because on a non-ECS or non-2.0 system, Hires-Interlace is the closest old viewmode available for attempting to display a Productivity-Interlace image.

By storing the entire 32-bit ModeID in the CAMG chunk under 2.0, new reader applications can attempt to redisplay the image in its intended display mode when possible (i.e., when running under 2.0 on a system capable of the display mode). Old readers will not be able to use the new display modes, but their existing code will truncate the new 32-bit ModeID into a 16-bit viewmode which will generally provide an old mode capable of displaying the image in some fashion.

New ILBM readers and readers revised for 2.0 can apply logic such as the following when supporting new display modes:

```
struct Screen *openscreen(ULONG modeid, SHORT wide, SHORT high, SHORT deep)
{
extern struct Library *GfxBase;
struct Screen *screen;

if(GfxBase->lib_Version >= 36)
{
/* if mode is not available, try a fallback mode */
if(ModeNotAvailable(modeid)) modeid = fallbackmode(modeid).

if(!(ModeNotAvailable(modeid))) /* if mode is available */
{
/* We have an available mode id
Here you may wish to create a custom, or centered, or overscan
display clip based on the size of the image. Or just use
one of the standard clips.

The 2.0 Display program uses QueryOverscan to get the settings
of this modeid's OSCAN_TEXT, OSCAN_STANDARD, and OSCAN_MAX.
Display centers the screen (via TopEdge and LeftEdge) within
the user's OSCAN_STANDARD settings, and creates a display clip
by using the same values and then clipping the values to be
within OSCAN_MAX limits. If the centered screen ends up lower
than user's OSCAN_TEXT settings, I move it up to same MinY as his
OSCAN_TEXT --- otherwise his Workbench might peek over the top.
*/

/*
Now use extended OpenScreen or OpenScreenTags for this modeid.
(this gives you the benefit of system-supported overscan, etc.)
*/
}
}
}
```

```

if no display opened yet (either not 2.0 or mode not available or can't open)
{
    Try an old-style OpenScreen with NewScreen.Modes = mode & VPMODEMASK;
}

return(screen);
}

```

```

/*
 * fallbackmodeid - passed an unavailable modeid, attempts to provide an
 *                  available replacement modeid to use instead
 */

#define VPMODEMASK (~(GENLOCK_AUDIO|GENLOCK_VIDEO|VP_HIDE|SPRITES))

ULONG fallbackmodeid(ULONG modeid, SHORT wide, SHORT high, SHORT deep)
{
    extern struct Library *GfxBase;
    ULONG newmodeid;

    /* if it's an old 1.3-style mode, mask out inappropriate bits
     */
    if(!(modeid & MONITOR_ID_MASK)) newmodeid = modeid & VPMODEMASK;

    else newmodeid = modeid; /* else start with what was passed */

    if(GfxBase->lib_Version >= 36)
    {
        if(ModeNotAvailable(newmodeid))
        {
            /* Here you should either be asking the user what mode they want
             * OR searching the display database and choosing an appropriate
             * replacement mode based on what you or the user deem important
             * (colors, or aspect, or size, etc.). You could also use a built
             * in table for modes you know about, and substitute mode you wish
             * to use when the desired mode is not available.
             */

            newmodeid = ???
        }
    }

#ifdef DEBUG
    printf ("Trying 0x%08lx instead of 0x%08lx\n",newmodeid,modeid);
#endif
    return(newmodeid);
}

```

◆



Debugging Tools - Choosing the Right Tool for the Job

by Carolyn Scheppner

Note: Some of the debugging tools supplied on the Devcon disk are from the CATS support item "Software Toolkit". Full instructions on the use and options of some of the more complex tools such as Wack and Wedge may be found in the Software Toolkit manual.

General Warning: Some debugging tools stress the system, or allow you to wedge into arbitrary system routines, or attempt to provoke improperly written code to crash. We have attempted to mark these kinds of tools with warnings below. You probably should not write to un-backed-up disks or harddisks while using such tools.

Unless otherwise noted, the following tools are all Copyright (c) 1985,1990 Commodore-Amiga, Inc. All Rights Reserved. They are provided for debugging purposes only and may not be redistributed in any manner.

Watchdog Tools

Watchdog tools help trap illegal memory accesses. Such accesses are generally caused by using improperly initialized variables or structures, or by accessing structures and memory that have already been freed. Code with illegal accesses may appear to run fine under most circumstances but may fail or crash unexpectedly in the field.

Unfortunately, it is currently not possible to trap all illegal accesses. If a program is accessing or trashing memory in normal legal user memory spaces, or even trashing itself, these tools won't catch it in the act. Luckily, a majority of illegal accesses reference low memory or freed memory. By using a freed memory invalidation tool like *MemMung* in conjunction with an illegal access watchdog tool, the majority of these problems can be caught.

The best watchdog tools require an MMU. Processor-based tools such as *MemWatch* and *WatchMem* can watch for writes to low memory. But they can't catch reads of low memory or other illegal accesses.

New MMU-based watchdogs such as *Enforcer* and *CPU* can trap all illegal accesses of low memory, non-existent memory, and ROM, reporting the exact type of access, as well as the offending code's program counter and registers. The debugging information is sent to a serial terminal (or parallel printer with *CPU's cputrap.par*).

If the illegal access occurs in ROM code, you can generally trace forward on the stack to find the program address that called the ROM routine. It is then possible to disassemble a program in memory at the point it caused the illegal access. Programmers who like to debug at a low level may then either immediately recognize the problem, or can compare the code disassembled in memory to disassemblies of their object modules (or to their source code if the source is in assembler).

Programmers who prefer to debug at a higher level can compile a debugging version of their software to allow them to track which code is executing when the illegal access occurs. This can be accomplished by stepping or breakpointing with a debugger, or by inserting remote debugging statements (kprintf() or dprintf()) to the same remote device that is receiving the watchdog output. Plain printf() debugging could also be used with Delay()'s to allow time for watching both the printf() debugging and the remote watchdog output.

All software should be tested with a memory invalidator, such as *MemMung*, running in conjunction with one of the illegal access trappers. It is extremely useful to use such tools *while* you are developing so that you can catch illegal accesses right away - they are much easier to find without disassembly if you just wrote or changed the code.

TOOLNAME: Enforcer
CATEGORY: MMU-based Watchdog tool
USAGE: Enforcer on | off [parallel]
USED FOR: Trapping reads and writes of low/non-existent memory
REQUIRES: MMU that is not being used, serial terminal or parallel printer
FOUND ON: Devcon disk

TOOLNAME: Cpu/Cputrap (and Cputrap.par)
CATEGORY: CPU/MMU utility with auxiliary watchdog trap handlers
USAGE: RUN cputrap (or cputrap.par), then CPU TRAP
USED FOR: Trapping reads and writes of low/non-existent memory
REQUIRES: MMU that is not being used, serial terminal or parallel printer
FOUND ON: Workbench 2.0 (trap handlers on Devcon disk)

TOOLNAME: Lawbreaker
CATEGORY: Test program for Enforcer or CPU/Cputrap
USAGE: Lawbreaker
REQUIRES: CPU or Enforcer
FOUND ON: Devcon disk

TOOLNAME: WatchMem (inspired by MemWatch by John Toebes VIII)
CATEGORY: Processor-based low memory watchdog
USAGE: RUN Watchmem [file | window] opt n [interval] (opt n = nocorrect)
USED FOR: Trapping writes to low memory
WARNINGS: This processor-based tool can not prevent writes to low memory. It can correct them after they occur, but you might crash first.
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: Complainer
CATEGORY: Watchdog trap for A3000
USAGE: Complainer ON/OFF (Use with MemMung)
REQUIRES: Serial terminal
USED FOR: Monitoring accesses of non-existent memory (bus errors)
FOUND ON: Devcon disk

TOOLNAME: MemMung
CATEGORY: Memory invalidation tool (more pleasant with Enforcer/CPU)
USAGE: RUN MemMung
USED FOR: Catching accesses of uninitialized and freed memory
WARNINGS: Will provoke bad code to crash if not used with Enforcer/CPU
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: IO_Torture
CATEGORY: Specialized watchdog for IORequest re-use
USAGE: IO_Torture
USED FOR: Remote monitoring of premature re-use of IORequests
REQUIRES: Serial terminal
FOUND ON: Devcon disk

TOOLNAME: Codewatcher by Michael Plitkins
CATEGORY: Application resource allocation/deallocation watchdog
USED FOR: Checking if an application frees its resources properly
FOUND ON: BIX (?)

Monitoring Tools

TOOLNAME: Tstat
CATEGORY: Task monitor
USAGE: Tstat [CLI# | ExecTaskName] [-tickdelay]
USED FOR: Monitoring PC, regs, stack, signals, etc. of a running task
REQUIRES: Local monitoring is default, serial is optional
FOUND ON: Devcon disk

TOOLNAME: Wedge
CATEGORY: System function monitor
USAGE: Complex and best done with scripts - type Wedge help for help
USED FOR: Monitoring the calls to and results from any system function
REQUIRES: Limited local monitoring, serial or parallel for full monitoring
WARNINGS: Can bog system down; can crash if calling task has tiny stack.
Local monitoring can cause recursive looping if functions called
by text output routines are wedged.
FOUND ON: Devcon disk, Software Toolkit (full instructions with ToolKit)

TOOLNAME: DevMon (see DEVICE TOOLS)

Crash Trapping Tools

TOOLNAME: SRT
CATEGORY: Software error trapping wedge (for 1.2/1.3, not 2.0)
USAGE: SRT [srt.textfile] (default s:srt.text)
USED FOR: Examining name, registers, PC, SP, of crashed task
FOUND ON: Devcon Disk

TOOLNAME: TNT
CATEGORY: Software error trap handler (for all version of OS)
USAGE: TNT (must be installed before the crash occurs)
USED FOR: Examining name, registers, PC, SP, of crashed task
WARNINGS: You may need to do TNT OFF before using a trap-based debugger.
FOUND ON: Devcon disk

General Debuggers and Disassemblers

Many development language packages come with excellent source level debuggers and object module disassemblers. In addition, the following tools are useful for debugging executables:

TOOLNAME: Wack (Originated by Carl Sassenrath)
CATEGORY: Symbolic debugger/disassembler
USAGE: Wack "program [programargs]" (see SW Toolkit for other opts)
USED FOR: Disassembling, single stepping, breakpointing
WARNINGS: Improper use could lead to a crash. Wack1.0 installs/leaves a trap handler. If used with TNT, RUN Wack so only the handler of the bg run process will be changed.
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: RomWack
CATEGORY: ROM-based debugger
USAGE: Enter with exec Debug() function or RomWack command (SW Toolkit)
USED FOR: Freezing the Amiga while you examine memory remotely
REQUIRES: Serial terminal
WARNINGS: Improper use could lead to a crash.
FOUND IN: the Amiga OS

TOOLNAME: Metascope (by Metadigm)
CATEGORY: Multiwindow Intuition interface symbolic debugger/disassembler
USED FOR: Disassembling, single stepping, breakpointing
WARNINGS: Improper use could lead to a crash.
FOUND IN: Stores (Commercial product)

System Configuration Listers

Configuration listers are handy for checking the address, version, or presence of various system hardware and software items. If you are working with devices or libraries, you can use the memory tool *Flush* to flush your device or library from memory and *LibList* or *DevList* will check that the device or library has actually been removed from the system. *Config* can be used to check a machine's OS and custom chip versions, processor type, and configured devices without taking off the cover.

TOOLNAME: Config
CATEGORY: Motherboard and Autoconfig configuration lister
USAGE: Config [debug]
USED FOR: Checking ROM/Processor/Chip versions, and autoconfig devices
FOUND ON: Devcon disk

TOOLNAME: TaskList, LibList, DevList, ModList, (C. Sassenrath) IntList
CATEGORY: System software list display tools
USAGE: No arguments for any of these
USED FOR: Checking address, version, presence, of tasks, libs, devs, etc.
FOUND ON: Devcon disk (see also Memory Tool "Flush")

TOOLNAME: MemList - see MEMORY TOOLS

TOOLNAME: DosList - see DOS/DISK TOOLS

Memory Tools

Most memory tools are used to check for, and debug memory losses and other memory allocation and deallocation problems. *Avail* and *Flush* can be used together to make sure that an application is freeing all of its memory. *Flush* is required because libraries, devices, and fonts loaded from disk will hang around in memory even after they have been closed until someone asks for the memory.

To check your application for memory loss, arrange your Workbench so that you have an open shell (for *Avail*) and can start your application from a different shell or from an icon without rearranging any windows (rearranging windows causes memory fluctuations). If possible, size the shell window for *Avail* tall enough for the output of two avails and a couple of flushes (so that you won't have to write down any numbers).

Then, without rearranging any windows, do:

1. Flush
2. Avail (note these pre-application Available totals)
3. Start your application
4. [optional Avail here to check run-time memory usage]
5. Exit your application
6. Flush
7. Avail (the Available totals should match the pre-application ones)

TOOLNAME: Avail
CATEGORY: Memory free/largest lister
USAGE: Avail (2.0 has flush opt; use Flush command with earlier Avails)
USED FOR: Checking memory usage, and memory loss in conjunction with Flush
FOUND ON: Workbench

TOOLNAME: Flush
CATEGORY: Memory flusher (to check for real memory loss)
USAGE: Flush (Note - Flush does 3 flushes when invoked)
USED FOR: Flushing all currently unused devices/libraries/fonts from memory
FOUND ON: Devcon disk (use in conjunction with Avail)

TOOLNAME: MemMon
CATEGORY: Memory use recorder (helps narrow search for lost memory)
USAGE: MemMon (>diskfile)
USED FOR: Producing a commented record of memory usage
FOUND ON: Devcon disk
TOOLNAME: Frags
CATEGORY: Memory fragmentation summarizer
USAGE: Frags [full]
USED FOR: Checking for memory fragmentation.
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: MemWall
CATEGORY: Memory allocation overwrite/underwrite monitor
USAGE: Memwall [all] [fill N] [presize N] [postsize N] [snoop] [supersnoop]
USED FOR: Finding things that write outside their allocated memory
 Also for general snooping of memory allocations.
REQUIRES: Serial terminal
WARNINGS: Some things in the system (such as layers) free memory in smaller chunks
 than they allocated. When this is done, (or when it finds a fill area hit),
 it does NOT let that area actually be deallocated. This can lead to loss of
 memory. Note that presize or postsize may be 0.
FOUND ON: Devcon Disk

TOOLNAME: MemList
CATEGORY: Full used and free memory chunk lister
USAGE: Memlist [>diskfile]
USED FOR: Debugging fragmentation/deallocation problems
FOUND ON: Devcon disk, Extras(?)

TOOLNAME: Owner
CATEGORY: Memory ownership tool
USAGE: owner [0x] nnnn... (owner ? for help)
USED FOR: Trying to determine ownership of allocated memory
FOUND ON: DevCon disk

TOOLNAME: Snoop
CATEGORY: Remote AllocMem/FreeMem debugger
USAGE: Snoop (use SnoopStrip on captured output to isolate unfreed Allocs)
USED FOR: Debugging unfreed memory problems
REQUIRES: serial terminal
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: SnoopStrip
USAGE: SnoopStrip [>outfile] infile
USED FOR: Stripping matched allocs/frees from captured snoop output
FOUND ON: Devcon disk

TOOLNAME: Drip
CATEGORY: Memory loss accumulator
USAGE: Drip [threshold]
USED FOR: determining change in free memory since last invocation
FOUND ON: Devcon disk, Software Toolkit

TOOLNAME: Peek
USAGE: Peek B|W|L [0x]address [[0x]compvalue] [[0x]mask]
USED FOR: Checking or script branching on contents of a memory address
FOUND ON: Devcon disk

TOOLNAME: Poke
USAGE: Peek B|W|L [0x]address [0x]value [[0x]mask]
USED FOR: Changing the contents of a memory address
WARNINGS: Obviously, poking where you shouldn't may crash machine.
FOUND ON: Devcon disk

DOS/Disk Checking Tools

TOOLNAME: DosList
CATEGORY: Dos device lister
USAGE: Doslist [DEVS|VOLS|DIRS]
USED FOR: Examining the dos device list
FOUND ON: Devcon disk

TOOLNAME: ShowLocks (Copyright 1988 Chuck McManis)
CATEGORY: Filelock lister
USAGE: ShowLocks [volumename:]
USED FOR: Displaying outstanding locks
FOUND ON: Software Toolkit

TOOLNAME: DiskEd
CATEGORY: Disk sector editor
USAGE: DiskEd drivename: (see Bantam AmigaDos manual for instructions)
USED FOR: Examining and modifying disk sector data
WARNINGS: Improper use can trash disk data or structure.
FOUND ON: Devcon Disk

Performance Checking Tools

TOOLNAME: PerfMon
CATEGORY: System performance monitor
USAGE: Perfmon
USED FOR: Checking for busy waiting and other performance problems
FOUND ON: Devcon Disk, 1.3 Extras

Intuition/Graphics Tools

TOOLNAME: WinList
USED FOR: Examining addresses, titles, flags, sizes of screens and windows
FOUND ON: Devcon disk

TOOLNAME: ShowGfxBase
USED FOR: Examining GfxBase normal display sizes and flags
FOUND ON: Devcon disk

TOOLNAME: ReadPixel
USED FOR: Reading the XY screen location and color of pixels. Can be
used to check the size and position of onscreen images.
USAGE: Readpixel (then click on pixels to read)
FOUND ON: Devcon disk

Device Tools

TOOLNAME: DevMon
CATEGORY: Device monitor
USAGE: Devmon name.device unitnum [remote] [hex] [allunits] [full]
(remote is serial output, full has exec wedges in DoIO, ReplyMsg)
USED FOR: Monitoring the calls to a device
REQUIRES: Nothing for local monitoring. Serial terminal optional (slower)
WARNINGS: Stresses system if wedged into high-usage or time-critical devices
(such stress could lead to crashes or hangs)
FOUND ON: Devcon disk

TOOLNAME: Cmd
CATEGORY: Parallel/Serial output capture tool
USAGE: Type cmd help for usage
USED FOR: Debugging printer, serial, and parallel output
FOUND ON: Workbench 1.3

TOOLNAME: PrinterTest
CATEGORY: Printer driver test suite
USAGE: Printertest (then answer y when the correct printer is prompted)
USED FOR: Testing printer drivers
FOUND ON: Devcon disk

TOOLNAME: IO_Torture (see WATCHDOG TOOLS)

Development Time Bug Prevention/Tracking Tools

TOOLNAME: RCS
CATEGORY: Source/document control
USAGE: See accompanying docs on Fish Disk 282
USED FOR: Recording changes to source code and documents
FOUND ON: Fish Disk 282

TOOLNAME: Autodoc
CATEGORY: Source code autodoc extractor
USAGE: See accompanying autodoc.doc
USED FOR: Extracting standard function documentation from your source code
FOUND ON: Devcon disk

TOOLNAME: Bumprev
CATEGORY: Revision bumper
USAGE: Bumprev version revname (example: bumprev 36 my_rev)
USED FOR: Updating revision include files (xxx_rev.h and xxx_rev.i)
FOUND ON: Devcon disk



Debugging Amiga Software

by Carolyn Scheppner

This article presents some general techniques for debugging software on the Amiga. Before you start programming the Amiga, it's a good idea to read the development guidelines in the introduction of the the Addison-Wesley Hardware Manual (ISBN 0-201-18157-6) or ROM Kernel Reference Manual: Libraries and Devices (ISBN 0-201-18187-8). These guidelines contain important rules which are applicable to all Amiga programs, configurations, and operating system releases. Additional information can be found in the Troubleshooting Guide published in Amiga Mail and in the Libraries and Devices manual. These documents cover the most common Amiga programming problems.

Preventing Bugs

The best way to debug software is to prevent bugs in the first place. Accordingly, here are seven basic rules you should always follow when writing Amiga software:

1. Read the latest autodocs and the include file comments for the functions and structures you are using.
2. Always check return values from system functions. Provide a clean way out and useful messages if something fails!
3. Assembler programmers - remember to TST.L D0 after system calls, before branching on condition codes.
4. C Programmers - Use function prototypes for system functions and your own functions. It's a little extra work but will save you time in the long run by immediately catching most types of improper function calls (missing arguments, swapped args, etc.)
5. Keep a version number in your code and update the version number whenever changes are made. The 2.0 VERSION command can print out the version of any executable which contains a specially formatted version string:

```
In C:      UBYTE *vers="\0$VER: programname 36.10";
In Asm:    vers DC.B 0,'$VER: programname 36.10',0
```

6. Document the code changes for each version. This can be done manually or by using a document control system such as RCS.
7. Test your code! Test on different configurations, under low memory and error conditions, and in conjunction with various watchdog tools. Test your product with MemMung, if possible in conjunction with Enforcer or CPU/Cputrap (on A2500) or Complainer (on A3000) to catch uses of null pointers and freed memory.

Finding and Fixing Bugs

It is hard to generalize about debugging because different kinds of bugs often require very different approaches. A bug report from a user is quite different from a bug that you've just introduced in new code! However, all debugging requires some common steps:

1. Define the problem
2. Narrow the search and find the bug
3. Understand, and fix the bug
4. Make sure you didn't just break something else

Steps 3 and 4 are the same for all types of bugs, so we'll cover those last. Steps 1 and 2 require different approaches for different kinds of bugs. Here are some examples.

You've added or written new code and something is broken.

1. Define the problem. Make sure you can reproduce the problem so you'll know when it's gone. Define as "when I do xxx the program does (or doesn't do) do yyy."

2. Narrow the Search. If you just added a couple of lines of code, and have the same development environment as before, check your source code first. Check for misuse of existing variables, improper error checking, improper use of system or internal functions, and possible changes to conditional program flow.

If you can't spot the problem, it's time to slow it down and see what's going on. Use a source level or symbolic debugger, or print/kprint/dprint debugging, with delays added if necessary. One particularly useful type of debugging statement is:

```
printf("About to do xxx. k =%ld Ptr1=%%lx...\n",k,Ptr1);  
Delay(50);
```

The delay gives the debugging line time to be output and gives you a chance to read it before the action is taken. See `mydebug.h` on the 1990 DevCon disks for easy ways to add conditional debugging statements like this to your code.

By stepping through or printing out your actions and variables, you will generally be able to isolate the bug. If you have isolated the area but still can't find the bug, re-read the autodocs for the routines you are using. Check the Troubleshooting Guide in the Addison-Wesley *Libraries and Devices* manual. Check all other uses of the variables in the problem area. If all else fails, isolate the problem code by writing the smallest possible example that demonstrates the problem.

If the problem is not present in the smallest possible example, then go back and check your code. If the problem is still present, contact CATS for assistance or upload the example to BIX (note - one of the quickest ways to find bugs in a small source code example is to upload the source to BIX amiga.dev/main and ask what's wrong with it).

B. Your code has intermittent problems that you can't pin down, or appears to trash something under certain conditions.

1. Define the problem. It is difficult to reproduce intermittent problems, so try to force the problem to show itself. First try running your program with MemMung and one of the MMU watchdog tools. If you don't have an MMU, use WatchMem and MemMung, but be prepared to crash a lot. If you don't get any hits, try the same thing during low-memory situations, heavy multitasking and device IO, etc. If you are doing Exec device IO, try IO_Torture to catch premature reuse of IORequests. Hopefully, you will pick up a hit.

2. Narrow the Search. If you have no MemMung/Enforcer hits, try some debugging statements or source level debugging to follow the values of your variables. Use TStat to see if your stack usage is high. Check all possible areas where you might be overwriting the end of an array or otherwise trashing memory. Re-read the autodocs for the system functions you are using.

If you are reusing an IORequest too soon, check your source code (debugging would just slow down your execution and might give the IORequest a chance to complete, masking the problem).

If you have Enforcer hits, use debugging statements or a debugger to step through your code WHILE running MemMung and an MMU watchdog tool (or WatchMem). This will allow you to pinpoint where the problem occurs.

C. Your code works fine on one system but not on another. Or you've received a bug report from a user.

1. Define the problem. First find out the exact configuration of the system the problem occurred on. Important elements include memory configuration and addresses, amount of free Chip and Fast RAM, processor type, custom chip version, expansion peripherals, OS version, and other software in use when the problem occurred. The Config program on the 1990 DevCon disks is useful for printing out much of this information.

The memory address ranges can be particularly important now that machines are available with memory beyond the 24 bit address limit. For example, overwriting a byte array by one byte now has a good chance of trashing a 32-bit address variable, or even your routine's return address on the stack.

If a user reports the problem, find out the exact version of your software they are running, how they launched the program, and what their stack is set to (if launched from CLI). Try to get them to reproduce the problem in a known environment (ie. after booting with a release Workbench diskette). Get their phone number and keep it with a record of all of the information you can get on the problem. Keep bug reports in an organized form. If you get two reports on the same problem, you can be pretty sure that the problem really exists, and the combined information may help you track it down.

2. Narrow the Search. Attempt to reproduce the problem. If you can't reproduce it immediately, try stepping through the problem area while using MemMung and a watchdog tool. If you don't get any hits, try again with less memory available and other tasks running. Try to reproduce the user's configuration and environment. If you still can not reproduce the problem, ask the user to come up with a simple repeatable sequence which causes the problem on a system booted with a normal release Workbench disk.

Read the Troubleshooting guide in the Addison-Wesley *Libraries and Devices* manual or *Amiga Mail Technotes* for information on the causes for many problems that only show up in certain configurations or environments.

If all else fails, look carefully at your code for misuse of variables or system functions, and for improper error-checking or cleanup after any allocation or open. Check that all cleanups are done in the proper order.

D. Your program loses memory.

1. Define the problem. First make sure that you are actually losing memory. Use Flush (from the 1990 DevCon disks) and Avail to check for actual memory loss.

Set up your system so you have a shell window available *and* can start your program without moving any windows (re-arranging windows causes memory fluctuations). Test for memory loss as follows. First, try Flush and Avail a few times to make sure nothing else in your system is causing memory to fluctuate. Then perform the following steps.

1. FLUSH
2. AVAIL (write down the Fast, Chip, and total memory free)
3. Start your program and use its features
4. Exit your program
5. FLUSH
6. AVAIL (compare the fast, chip, and total free to previous figures)
7. If you have a loss, go back to step 2.

Two problems which show no net loss after exit but cause memory to be used up while you are running are:

- ☐ Opening diskfonts repeatedly
- ☐ Not keeping up with IntuiMessages

If you are opening diskfonts, first try OpenFont() in case the font is already in memory. If OpenFont() succeeds, check the size and flags against what you asked for. If it is correct, use it. Else close it and do OpenDiskFont() instead. If you are asking for voluminous IntuiMessages such as MOUSEMOVE, you must keep up with them. If you don't, Intuition will continue to allocate new blocks of messages and will not free them until you close the window.

2. Narrow the search. Try the above test again, but this time just start your program and exit immediately. If you do not lose memory, try several times more, using some of your program's features, and attempt to determine which part of your program causes the memory loss. Check your source code for all opens and allocations and check for matching frees and closes, in the proper order, for each of them.

The size of a memory loss can also be a clue to the cause. For example, a loss of exactly 24 bytes is probably a Lock() which has not been UnLock()'d. Knowing the exact size of the loss (as determined with Flush and Avail) is important when you try determine which allocation is not being freed.

Some additional tools on the 1990 DevCon disks can help determine where memory losses occur. You can use MemMon to record the relative memory usage as you test various parts of your program. Snoop can be used to record all memory allocations and frees on a remote terminal, after which SnoopStrip can strip out all matching pairs. MemList, which outputs the system memory list, can also be used to spot unfreed allocations.

The Wedge program, which can restrict its reporting to the function calls made by a single task or list of tasks, can be used to monitor the allocations and frees done by your task. By inserting debugging statements, you can mix status messages ("About to do xxx") with Wedge's output. Examine the output for an allocation which matches the size of your loss.

Removing Bugs

I mentioned steps 3 and 4 earlier. This is the easy part (finding the bug is the hard part). These steps are the same for most debugging problems.

3. Understand, and Fix the Bug. When you find the bug, make sure you understand it. Don't just try something else. If you are having a problem with a system routine, read the autodocs and chapter text for that routine. Check the Troubleshooting Guide in the 1.3 Addison-Wesley *Libraries and Devices* manual.

When you understand what is wrong, fix the problem, being especially careful not to affect the behavior of any other parts of your program. Carefully document the changes that you make and bump the revision number of the program. Note your changes in the initial comments of the program, and in the area where the changes were made.

4. Make Sure You Didn't Break Anything New. Try to reproduce the problem several times and make sure it is gone. Thoroughly test the rest of your program and make sure that nothing else has been broken by your fix. Test your program in combination with watchdog tools such as MemMung and Enforcer or Cpu/Cputrap.

Debugging Tools

1. NEW MMU Watchdog Tools: Enforcer, CPU/Cputrap, and Complainer

These new MMU-based Amiga debugging tools provide debugging and quality assurance capabilities far beyond what was previously possible. It is now possible to find bugs even in code that appears to be working perfectly - the kinds of bugs that could cause serious problems on different configurations. These tools are able to trap improper low memory accesses, writes to ROM, and accesses of non-existent memory - problems which are generally caused by use of freed or improperly initialized pointers or structures.

All software should be tested with these tools during development, and should be required to pass a test with an MMU watchdog in conjunction with MemMung and IO_Torture before being release and distributed.

2. Symbolic and source level debuggers

Symbolic debuggers allow you to trace and single step through your code, and examine or change your variables and structures. The source level debuggers which are provided with some compilers allow you to trace and single step your code at the source level after compiling with special flags. Debuggers can often be used in combination with other debugging tools such as MemMung and Enforcer to detect exactly where a problem is occurring.

3. Printf() and kprintf() / dprintf() debugging

This simple method of debugging allows you to monitor where you are, what your variables contain, and anything else you care to print out. Printf debugging is suitable for any process code that is not in a Forbid or Disable (printf breaks a Forbid or Disable). Kprintf (serial) and dprintf (parallel) debugging is more flexible and can be used in process, task, or interrupt code. The kprintf function is provided in the debug.lib linker library. The parallel version, dprintf, is provided in the ddebug.lib linker library. See the debug.lib kprintf autodocs for more information on the types of formats handled by kprintf and dprintf.

Kprintf outputs to the serial port at whatever baud rate the port is currently set to. Generally, kprintf is done at 9600 baud with a terminal, or another Amiga running a terminal package, connected to your serial port with a null modem serial cable.

However, it is possible to `kprintf` to yourself (ie. to a terminal package running on your own machine) if you have a modem attached to your serial port, and your terminal package set to the baud rate of your modem. Obviously, if the problem you are debugging causes you to crash, a remote terminal is a better choice. The ASCII capture feature of your terminal package can be used to capture the `kprintf` debugging output for later examination.

Remote (`kprintf`/`dprintf`) debugging is extremely useful when combined with other remote debugging tools such as Enforcer and Cputrap because your own debugging statements will be interspersed with the remote output of the other debugging tools, allowing you to track what your program is doing when problems occur.

`Printf`/`kprintf`/`dprintf` debugging can be conditionally coded more conveniently by using an include file such as `mydebug.h` (see the DevCon disks). `Mydebug.h` eliminates the need for messy `#ifdef` and `#endif` lines around your debugging statements by providing the conditional macros `D(bug())`, `D2(bug())`, and `DQ(bug())` which take `printf`-style format strings and arguments in their inner parens. One handy feature of these macros is that your debugging statements can be quickly changed from `printf`'s to `kprintf`'s or `dprintf`'s by simply setting a flag in `mydebug.h` and recompiling.

Example: `D(bug("I'm here now and a=%ld",a));`

4. Other ways to debug low -level code

If you can't link with `debug.lib`, low level code can also be debugged by inserting visual or audio cues to let you know where you are. `DebTones.asm` (in AmigaMail and on the 1990 DevCon disks) demonstrates a small audio tone macro suitable for debugging low level code. Another common method is flashing the power LED (see `togl_led.asm`), or doing an `Intuition DisplayBeep()` to flash the screen.

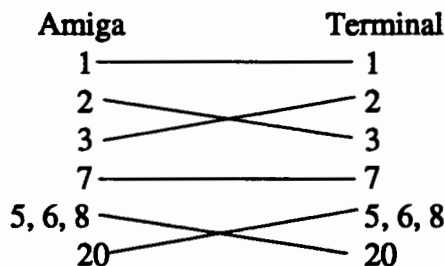
5. Specialized debugging tools

A variety of specialized debugging tools are available for monitoring and debugging such things as system function calls, device IO, process status, memory usage, and software errors. These tools can be used without recompiling your program and can provide valuable debugging information. See the list of tools accompanying this article in the DevCon notes.

How to Use MMU Watchdogs and Other Remote Debugging Tools

Remote Serial Debugging

Hardware Setup: When hooking two Amigas together, use a straight RS-232 cable with a null-modem adaptor, or use a null-modem cable. When hooking an Amiga up to another type of computer or terminal, you may or may not need the null-modem (crossed lines) depending on whether the other machine's RS-232 port is designed to be basically a sender or a receiver. Avoid connecting lines which are not directly related to RS-232 because different computers have various power supplies and grounds on these other lines. My null modem debugging cable is wired as follows:



Software: For remote debugging at 9600 baud, set the sending machine's Preferences to 9600 baud, and use a 9600 baud terminal or an Amiga running a 9600 baud terminal package (preferably with ASCII capture capability) as the receiving machine. Note that other baud rates can also be used for most serial debugging because normal serial kprintf's do not modify the serial SERPER register and are therefore output at the last baud rate your serial hardware was set to. Test your setup by copying a small text file to SER: or try the ktest program from the 1900 DevCon disks. The output should show up on the remote terminal.

Applications can output serial debugging statements by using kprintf from C or KPrintf from assembler and linking with amiga.lib and debug.lib. See the debug.lib autodocs for more information. Serial input functions are also available. Also see the mydebug.h conditional debugging macros on the 1990 DevCon disks.

Watchdog software setup: Make sure your test machine is set to the same baud rate as the remote terminal you are connected to. Turn on the ASCII capture of your remote terminal.

For A3000:

[RUN] MemMung (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
Complainer ON

For A2500:

[RUN] MemMung (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
Enforcer

For non-MMU machines (warning - encourages bad software to crash!)

[RUN] MemMung (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
[RUN] WatchMem

Setup for Local Serial Debugging

Hardware: If you have a modem attached to your serial port, it is possible to capture your own serial debugging output locally. This setup can be useful as long as the problem you are debugging is not one which crashes the machine.

Software: Run a terminal package at your modem's baud rate to capture the kprintfs. You probably won't be able to test this setup by copying a file to SER: (since the terminal package probably has an exclusive open on the serial device). Instead, use a small program like ktest (on the 1990 DevCon disks) to test your setup, or, if you already have an MMU watchdog installed, try an illegal memory accessor such as Lawbreaker. Use the terminal package's ASCII capture feature to capture your debugging output.

Watchdog software setup: Same as for remote serial debugging, but first start up a terminal package on the test machine, at the baud rate of the attached modem, with ASCII capture turned on.

Setup for Parallel Debugging

Hardware: To set up for parallel debugging, attach a parallel printer to the Amiga's parallel port and turn the printer on. Note - if no device is attached to the parallel port, parallel debugging statements will hang waiting for the port hardware.

Software: Some debugging commands have options for parallel rather than serial output. Examples include Cpu used with cputrap.par, and Wedge with the 'p' option. Also, you can send your own debugging statements to the parallel port by using dprintf from C, or DPutFmt from assembler, and linking with ddebug.lib and amiga.lib. On the 1990 DevCon disks, see dtest.asm for an example of calling DPutFmt from assembler, and mydebug.h for debugging macros which can use printf, kprintf, or dprintf.

Watchdog software setup:

For A3000:

[RUN] MemMung (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
Complainer.par ON

For A2500 running 2.0:

[RUN] MemMung (removable with CTRL-C, or BREAK n if RUN)
[RUN] IO_Torture
RUN cputrap.par
CPU trap



1

2

3



Ten Steps to Better Translations

by Dina Bennett

You've decided to sell your software overseas. And you've found, to your delight, that many of the marketing steps overseas are quite similar to what you've already done in the U.S.A. Sure, some of the parameters might be different. But basically you're still searching for distributors, negotiating contracts, tailoring a sales campaign . . . just like you do in this country.

Everything is so familiar that it's easy to overlook the one essential ingredient for successful sales overseas: translation. And then the day arrives when your distributor says, "By the way, we do expect your package to be in our native language."

All of a sudden you have to make major decisions about an unfamiliar issue. There's no time to lose and questions abound. What's involved in translation? Who should do it? How will we produce the foreign version? What will the project cost? When can it be delivered? And how will we know if the work is done well?

Few software firms have sufficient on-going translation volume to warrant doing it in-house. So the vast majority work with outside vendors who range from professionals to moonlighters, agencies to independent contractors. Figuring out who to use and what to expect is pretty tough, especially when you're not even sure what to ask for. Following is a brief primer on the ten essentials of internationalization. If you're not yet ready to sell overseas, use them to develop a proactive plan of action for the future. If translation is already a pressing issue, use them to make sure you have all your bases covered. And if you've already done translation, but you're not pleased with the outcome, use them to make sure things go better next time.

1. Evaluate your translation needs.

First determine what will be translated (manuals, software, training materials, sales pieces). Then decide whether you want each piece to match the look of the English original, which languages you need and when you need them. Your goal here is to figure out what you want done, regardless of who ultimately does the work.

2. Select a translation service.

You can get good recommendations from colleagues in other firms, or software organizations of which you are member. In deciding whether to use a freelancer or an agency, your three key selection criteria should be the vendor's direct experience, resources and range of service. Experience makes your vendor a leader and a problem-solver, and keeps you from being a guinea pig. Resource gives him the ability to absorb the changes in project scope that are inevitable when dealing with software. And range of service is helpful because you're likely to need such things as project engineering, desktop publishing and printing in addition to translation.

3. Plan the project.

Use your vendor's experience to help identify key components of the project. Assign clear responsibility for each part and notify all the parties of their role. Internationalization demands extra time and attention from everyone, above and beyond their normal work. If everyone agrees that internationalization is a company priority you will be able to set a realistic schedule for completion.

4. Structure the process for efficiency.

It's usually most practical to work with a vendor who can handle all the services you've identified in Step 1. If you use a separate agency for each language pair and service, you're multiplying the number of phone calls and crises you'll have to deal with. And you give up precious control over the quality and schedule of your project.

The more complex the project, the more it will benefit from having one key person in your company with overall responsibility for it. Your international products coordinator can be the conduit through whom all project information, questions and problems flow. While he may not yet have his own staff, he should have the authority to secure work from other departments, such as Engineering and Technical Publications. Your coordinator should have an equivalent on the vendor side who is aware of and facilitates every aspect of the project.

5. Confirm schedules.

Here is where good planning proves its value. From the basic project schedule you can break each component into its respective parts for tracking. There will be times, for example during documentation translation, when the translation firm needs little contact with you. But when foreign software is being tested, you may need to be in touch every day.

This is also the time to double-check that the schedule of those crucial to your project hasn't changed. Is the freeze date for U.S. software the same? Will your distributor still be available for reviews within the agreed upon time frame?

6. Confirm staffing.

After everyone has agreed on the target date for availability of foreign versions you can take a more realistic look at who's around to help with the project. This is the time to determine how much you can realistically handle in-house and still meet deadlines, as well as what remains to be contracted out.

7. Finalize costs.

Translation and production can be estimated quite precisely if you give the vendor sufficient materials on which to base his quote and if you can commit to certain assumptions and specifications. However, if your package isn't even in beta testing, you may have to be satisfied with an estimate that changes to a firm quote when U.S. software and documentation are frozen.

Don't neglect to figure your internal costs as well. You may find that it makes better sense to contract out more work than to add a staff position that costs you salary and benefits well after the project is finished.

8. Identify support resources within your company.

Inevitably there are times when an answer is needed that is crucial to the work. Or when a decision must be made concerning product issues impacted by translation. Setting up a team with key decision-makers from each department will help you respond quickly to your agency's requests. By identifying internal resources beforehand, you can easily turn to the right person for information. This takes pressure off of you and minimizes project delays.

9. Build quality assurance into the process.

Validation of software against documentation is even more important for foreign versions than for the original. In looking at your basic translation costs and your revised release schedule, you may be tempted to eliminate this quality control step from the process. Don't. Look at it this way: You'd never release a new package in the U.S. without checking it thoroughly. And a translation is akin to creating a whole new package. Validation may stretch the schedule some and it may add a bit to your total budget. But it pays you back tenfold in that the final product is every bit as good, and maybe even better, than the original.

10. Incorporate internationalization into future releases.

There is no doubt that internationalization can be done quicker, easier and at less cost when certain guidelines are followed during development. For example, translated text can be 20% to 30% longer than the original. But dialogue boxes, windows and packaging limited to the length of English text may not have room for the expanded text. So they have to be redesigned, adding to the cost of your foreign version. Software developed for the global marketplace includes translation requirements in its specifications and avoids these costly extras. In summary, select and work with your translation vendor as you would your law firm or ad agency. To paraphrase a line from the movie *Moonstruck*, "Good translation costs money. It costs money because it saves money." By viewing the relationship as long-term you'll find your initial investment returned each time there's a new release or a new language to translate.

Dina Bennett is vice-president of International Language Engineering Corp, 2540 Frontier Ave., #200, Boulder, CO 80301; Phone: (303) 447-2363. ILE specializes in software internationalization, their European headquarters is in France. ♦



VADs, VARs, DVARs and OEMs PROGRAM SUMMARY

Donna O'Neil

Technical Support Liaison, OEM/VAR Markets

VAD'S AND VAR'S

Proposal: This report outlines the proposed Value Added Sales Programs in relation to the existing Professional Dealer Channel.

The attached information covers:

- **Vertical Specialty Dealer Program (VAD)**
- **Value Added Reseller Program (VAR)**
- **DVAR (Dealer Value Added Reseller)**
- **VAR Application**
- **Reseller Agreement with VAR Schedules**

Since our current reseller program obligates the dealer to have a traditional storefront, these programs address resellers with specific industry expertise or those who manufacture a proprietary hardware / software but may not have a traditional retail environment.

VERTICAL SPECIALTY DEALER (VAD)

Definition: A Vertical Specialty Dealer is a reseller who has expertise and sells into a particular industry. These reseller may add value through software and/or hardware and sells integrated solutions to their customer base. Accounts will be handled through regions under normal reseller terms and conditions.

Examples:

- **Pro-Video Dealer** - sells high end video equipment to corporations, education, government, business markets.
- **Pro-Graphics/Art Dealer** - sells high-end graphics/art supplies to the creative arts market.
- **Pro-Music Dealer** - sells high end music equipment to the entertainment market.

Product Requirements: For Resellers with little knowledge of the computer industry, market specific bundled configurations of hardware and software will be offered. Those with expertise will be able to buy a limited product range.

<u>Vertical Specialty</u>	<u>Examples of Product Bundles</u>
Video	Video 1 and 2 Bundles with genlock added
Music	MIDI, A2000HD, Monitor, Software
Graphics/Arts	DPaint, Animate, Sculpt, Digitizer

Location: Storefront or Showroom Display for the display of demonstration equipment.

VERTICAL AUTHORIZED RESELLER (VAR)

Definition: A Vertical Authorized Reseller is a reseller who adds value to the Amiga product through integration of their proprietary software, hardware, or peripherals. The VAR sells their solution into a particular industry.

Market Examples:

- Laboratory Data Management (Pittsburgh)
- Heifner Communications (Satellite Communications)
- Interactive Video
- Computer Based Training

Product Requirements: Specific product required to fit total solution.

Location: Non Storefront

PROGRAM REQUIREMENTS

SERVICE SUPPORT: PRE-SALE :

- Technical support with corporate liason
- Scheduled Engineer Meeting at Corporate
- Service technical and service review

POST-SALE:

- Technical Bullentins
- Function, Form, Fit Changes
- Q & A Connection

Non-Disclosure - Beta Test new hardware/software

MARKETING SUPPORT:

- PR Opportunities
- Referrals
- Promotional Market Advertising
- Joint Marketing
- Trade Shows
- Direct Marketing

SALES SUPPORT:

- 3 VAR Salespersons from Corporate
- 1 VAR Support Person

LAUNCH: Press release to all appropriate media
Announce Paint Store solution and Laboratory Data
Management Systems, others.
Ad in VARBusiness

OTHER MARKET OPPORTUNITIES:

- Service - Customer Configuration
- 3rd party products for specific applications

Value Added Reseller LEVEL 1

Commitment:	100 - 500K
Product Pricing:	31 - 36% Amiga 39% PC Peripherals-Product Specific Quantity Discounts
Development Funds:	5%
Shipping:	VAR pays freight
Terms:	Standard with ability to adjust for project volume peaks
Demo Package:	3 / model maximum demos at 50% off Extended terms of 60-90 days with credit approval
Service:	Internal or 3rd Party
Training:	Hardware Technology Training Service Authorization (where applicable) Developer Authorization (where applicable)

Value Added Reseller LEVEL 2

Commitment:	500 - 1 million
Product Pricing:	32 - 37% Amiga 42% PC Peripherals-Product Specific Quantity Discounts
Development Funds:	5%
Shipping:	Free freight
Terms:	Standard with ability to adjust for project volume peaks
Demo Package:	4 / model maximum demos at 50% off Extended terms of 60-90 days with credit approval
Service:	Internal or 3rd Party
Training:	Hardware Technology Training Service Authorization (where applicable) Developer Authorization (where applicable)

Value Added Reseller LEVEL 3

Commitment:	1 million +
Product Pricing:	37 - 41 % Amiga 47% PC Peripherals-Product Specific Quantity Discounts
Development Funds:	5%
Shipping:	Free freight
Terms:	Standard with ability to adjust for project volume peaks
Demo Package:	6 / model maximum demos at 50 % off Extended terms of 60-90 days with credit approval
Service:	Internal or 3rd Party
Training:	Hardware Technology Training Service Authorization (where applicable) Developer Authorization (where applicable)

DEALER VALUE ADDED RESELLER (DVAR)

Definition: A small volume value added reseller who sells into a particular market and adds value with hardware or software applications.

Program: Would permit specific Authorized Professional Dealers to sell to approved DVAR's in their local market. Each DVAR would apply to Commodore Channel Management for Approval and be assigned to an Authorized Reseller. Legitimizes sales opportunity to small VAR.

The DVAR may purchase product directly from Commodore or from an approved aggregator. If the DVAR wishes to purchase directly from Commodore the terms are as follows:

Commitment:	Less than \$100,000
Product Pricing:	30 - 33% Amiga
Shipping:	Freight Collect
Terms:	Cash In Advance
Co-op:	No Co-op Funds, No MDF Funds
Service:	Although it is not mandatory for DVAR to be authorized service center, service arrangements are mandatory through the Authorized Reseller.
Minimum Order:	Minimum Order \$2500.

AGGREGATOR (AGAR)

Definition: An Authorized Commodore Reseller who agrees to provide product and specific services under the terms of an Aggregator Addendum to the Reseller Agreement. This addendum will specify by list any affiliated resellers and DVAR's.

Program: The Aggregator would purchase product under the normal price schedule and sell product only to the named affiliates and DVAR's under terms similar to the Commodore direct program.

Aggregator agrees to inventory 10% of his annual volume commitment.

Aggregator provides first line technical support to affiliates and DVARs.

Aggregator may extend credit terms and pass co-op through to resellers.

Aggregator will provide sales support documentation on a monthly basis or as required.

Rebate: As a result of providing the above services to the Affiliate / DVAR, Commodore will rebate the Aggregator 5% of Commodore invoice for any purchase of product for delivery to an Affiliate / DVAR.

ORIGINAL EQUIPMENT MANUFACTURER (OEM)

Definition: Manufacturer of specified products in which Commodore hardware components or assembled computer systems are integrated. The final product is no longer a Commodore product but a product of the manufacturer OEM.

Examples:

- Colwell General - Kiosk Display (ie. Paint Stores)
- Neilsen Kellerman - Animated Rowing Machine
- P.O.S.T. - In Store Product Advertising (Grocery Stores)

Product Requirements: Component level and assembled product to fit total solution.

Location: Non-storefront

PROGRAM REQUIREMENTS

SERVICE SUPPORT: PRESALE:

- Technical Support with Inside Liason**
- Scheduled Developers Meeting with CATS**
- Scheduled Engineering Meeting with Corporate**
- Service Specifications Review**

POST SALE:

- Technical Bullentins**
- Developer Update**
- Function, Form, Fit Changes**
- Internal Technical Liason**

NON-DISCLOSURE - Beta Test new hardware / software

MARKETING SUPPORT:

- PR Opportunities**
- Joint Marketing**
- Trade Shows**
- International Marketing**

SALES SUPPORT:

- 1 Sales / Technical Liason from Corporate**

LAUNCH:

- Press Release to all appropriate media**
- Announce at DEVCON, June 27th - 30th**

Commodore Business Machines

Professional VAR Application

May 29, 1990

The business entity which will own and operate the Commodore Business Machines authorization must complete this application.

Complete SECTIONS 1 through 3 and return to your Regional Sales Office or to the following address:

SECTION 2 must be completed for each location for which you are seeking authorization. Photocopies of SECTION 2 may be made for this purpose.

SECTION 1:
APPLICANT PROFILE

LEGAL BUSINESS NAME: _____

D/B/A: _____

BUSINESS ADDRESS: _____

DATE BUSINESS ESTABLISHED: _____ **FISCAL YEAR ENDS:** _____

PRIMARY CONTACT: _____ **TELEPHONE #** _____

CHECK ONE: ☐ Corporation ☐ Sole Proprietorship
 ☐ Partnership ☐ Sub-Chapter S Corporation

PLEASE ATTACH THE FOLLOWING:

- Corporate Organization Chart
- Description of other affiliations to include parent, subsidiaries, divisions, and other company relationships
- List of principal names and addresses

PLEASE COMPLETE THE FOLLOWING QUESTIONS:

A. Have you or a principal of your organization applied for CBM VAR status before?

_____ **Yes (Explain)**

_____ **No**

Please describe any current or previous relationship that your company has had with Commodore. _____

B. Which of the following best describes your business in the location(s) intended for this application: (Please check those that apply)

- | | | |
|--|---|--|
| <input type="checkbox"/> Video VAR | <input type="checkbox"/> Courseware Publisher | <input type="checkbox"/> Video Production House |
| <input type="checkbox"/> Computer Based Training | <input type="checkbox"/> Music VAR | <input type="checkbox"/> Third Party Maintainer |
| <input type="checkbox"/> Systems Integrator | <input type="checkbox"/> Mail Order Business | <input type="checkbox"/> Interactive Video Developer |
| <input type="checkbox"/> Graphics Design | <input type="checkbox"/> CAD/CAM | <input type="checkbox"/> Animation |
| <input type="checkbox"/> Other Specify: _____ | | |

C. Revenue Information: Please provide the dollar revenue by source for the indicated period:

	Preceding 12 Months	Next 12 Months
• Computer Hardware \$	_____	_____
• Software \$	_____	_____
• Supplies \$	_____	_____
• Consulting \$	_____	_____
• Service \$	_____	_____
• Education / Support \$	_____	_____
• Other _____: \$	_____	_____
• Total Revenue \$	_____	_____

D. What is your current product line? Please provide your gross annual sales, actual and projected by line:

	Preceding 12 Months	Next 12 Months
• CBM (if applicable) \$	_____	_____
• _____	_____	_____
• _____	_____	_____
• _____	_____	_____

E. If you currently sell Commodore products please report your unit sales for the period indicated and a projection for the 12 months following authorization:

	Preceding 12 Months	Next 12 Months
• PC 10/20	_____	_____
• PC 40	_____	_____
• PC 60	_____	_____
• A500	_____	_____
• A2000	_____	_____
• A2500	_____	_____
• A3000	_____	_____

F. How many personal computer systems did / will you sell in the following price ranges?

	Preceding 12 Months	Next 12 Months
• less than \$1599	_____	_____
• \$1600 - \$2999	_____	_____
• \$3000 - \$6499	_____	_____
• \$6500 - \$9999	_____	_____
• \$10,000 +	_____	_____

G. As a company, what is your annual advertising budget for personal computer and related products expressed as a percentage of gross sales? _____

How is this annual budget allocated? _____ Print Ads _____ Radio _____ Television

_____ Trade Shows _____ Yellow Pages _____ Other Specify: _____

Please include samples of current advertising.

H. With which of the following do you have official and authorized reseller status?

<input type="checkbox"/> CBM	<input type="checkbox"/> IBM	<input type="checkbox"/> DEC
<input type="checkbox"/> Apple	<input type="checkbox"/> NEC	Others: _____
<input type="checkbox"/> Compaq	<input type="checkbox"/> Sun Microsystems	_____
<input type="checkbox"/> Epson	<input type="checkbox"/> AST	_____

I. Describe your specific industry or vertical market expertise.

	Industry/ Vertical Application	% of Total Sales
•	_____	_____
•	_____	_____
•	_____	_____

What is your company's VALUE -ADD? (Please be as specific as is possible)

Please include any brochures that describes your VALUE -ADD.

J. How many VALUE ADDED systems do you sell per month from all applicable locations? _____

K. How would you describe your sales, service, and support territory?

☐ Local ☐ Regional ☐ National ☐ International

Please describe both the geographic and demographic variables of your target market(s). _____

L. How are sales distributed by target market?

Consumer _____

Business to Business _____

Federal/State/Local Govt. _____

Primary/Secondary Education _____

College/University Education _____

100%

M. Please provide the following:

- Corporate Sales and Marketing Plan
- A list of all locations that will be included in this application for professional VAR authorization, by name, if applicable, city and state. Please complete SECTION 2 for each location for which application is being made.

N. How do you sell your products?

- Inside Sales Person _____ %
- Outside Sales Person _____ %
- Mail Order _____ %
- Other specify: _____ %

O. Does your company offer warranty service and support for its existing product lines?

____ Yes ____ No At all locations? ____ Yes ____ No

If yes, do you stock maintenance parts for service? ____ Yes ____ No

If no, how will you maintain the Commodore product line?

3rd Party _____ Whom? _____

P. Do you offer post/ warranty service support for your existing product line?

SECTION 2:

LOCATION BUSINESS / MARKETING PROFILE

(Please complete the following form for each location)

LOCATION NAME: _____

ADDRESS: _____

LOCATION MANAGER: _____

DATE OPENED: _____ **TELEPHONE #** _____

PLEASE COMPLETE THE FOLLOWING QUESTIONS:

A. Locations Specifications: (check all that apply)

____ Urban	____ Suburban	____ Rural
____ Bus. District	____ Indust. Park	____ Strip Center
____ Stand-alone	____ Shopping Mall	____ Residential
____ Art Store	____ Music Store	____ Others (specify) _____

B. What is the percentage of location sales achieved in each market radius? (Complete both)

25 Miles account for ____ % of location sales

100 Miles account for ____ % of location sales

C. How would you characterize the business area served by this location?

____ Local ____ National ____ Regional ____ International

D. Please enclose a map of the area of this location and identify the location. Also designate:

- Locate sales territory
- Location of other VARS in the area in similar industries

E. Describe the square footage distribution of the area occupied at this location:

- Sales/demo area _____ square feet
- Service area _____ square feet
- Training/classroom _____ square feet
- Other:specify _____ square feet
- Total Square Feet _____ square feet

Please provide the following location photographs: (photos or VHS videotape)

***Sales/demo area**

***Service area**

*** Storage area**

***Exterior (Frontage and adjacent buildings)**

H. How many employees are supported from this location?

	Total	Full Time	Part Time
• Inside sales reps	_____	_____	_____
• Outside sales reps	_____	_____	_____
• Training Personnel	_____	_____	_____
• Programmers	_____	_____	_____
• Hardware Engineers	_____	_____	_____
• Tech Support	_____	_____	_____
• Others:specify:	_____	_____	_____
• TOTAL	_____	_____	_____

Describe the specialized staff support for your VALUE - ADD?

K. Are education classes and seminars provided to customers at this location?

_____ Yes

_____ No

SECTION 3

FINANCIAL AND OTHER INFORMATION

In order to comply with this section, please provide the following documents and/or information:

- Two most recent audited income statements and balance sheets
- An Annual Report, if applicable
- Bank references
- Other credit references, as appropriate

This application consists of SECTIONS 1 through 3. If you did not complete a section or sub-section please explain:

I have completed the above application and understand that the information provided is being used by Commodore Business Machines to evaluate my application for VAR status. The information I have provided is accurate to the best of my knowledge and belief.

By : _____ Date: _____

Signature: _____

Title (Corporate Officer): _____

AUTHORIZED RESELLER AGREEMENT

I. APPOINTMENT

Commodore Business Machines, Inc. ("COMMODORE"), hereby appoints the undersigned RESELLER ("RESELLER") and RESELLER hereby accepts appointment as a non-exclusive, authorized COMMODORE RESELLER for the direct sale of those products described in attached Schedule A ("Products"), to those customers described in attached Schedule B ("Customers"), at the location(s) described in the attached Schedule C ("Location"), subject to the Policies and Programs described in attached Schedule D.

II. TERM

Except as provided in Article IX, this Agreement shall commence on the date hereof and terminate one (1) year thereafter, and may be extended for subsequent one year periods upon the mutual written agreement of the parties.

III. CONDUCT OF RESELLER BUSINESS

A. RESELLER shall use its best efforts to encourage and develop the full retail sales potential for the Products, and to develop and maintain the reputation and goodwill of COMMODORE and the Products.

B. RESELLER shall conduct its business in its own name and at its own expense and risk, and shall be responsible, as an independent contractor, for compliance with all laws and regulations governing the conduct of its business.

C. RESELLER shall provide (i) full in-house support/service for Products, including but not limited to, the service and repairs covered by COMMODORE's consumer warranty; (ii) shall employ a reasonable number of personnel who have qualified as competent, trained and knowledgeable in the Products and the service and support of Customers; and (iii) maintain a suitable facility and sufficient inventory of Products and spare parts to fulfill its Customer requirements. COMMODORE will cooperate with RESELLER by making available to RESELLER reasonable training and materials related to such product support/service.

D. RESELLER shall provide COMMODORE with such periodic written reports and forecasts concerning the Products, market conditions and Customers as may from time to time be reasonably requested by COMMODORE.

IV. PURCHASES OF PRODUCTS

A. COMMODORE shall sell the Products to RESELLER, FOB, COMMODORE's distribution point, in accordance with COMMODORE's then applicable prices, terms and conditions, which COMMODORE may change from time to time. COMMODORE shall, however, provide at least thirty (30) days prior written notice of any price increase hereunder. Title to all Products purchased by RESELLER and all risk of loss or damage, in transit or otherwise, shall pass to RESELLER when the Products are delivered to the carrier or, if RESELLER has requested a delay in shipment, when placed at RESELLER's disposal as determined by COMMODORE. Unless specifically stated on COMMODORE's price list then in effect, the established Product prices do not include transportation, insurance, duty or taxes of any nature, all of such amounts being directly payable by RESELLER.

B. All orders for Products shall be ninety (90) day advance, firm non-cancelable orders which shall become binding upon acceptance in writing by COMMODORE or by COMMODORE's shipment of the Products ordered. COMMODORE reserves the right to reject any order hereunder. Furthermore, COMMODORE may cancel all or part of accepted orders or refuse or delay shipment of Products if COMMODORE determines, in its sole discretion, that (i) RESELLER has exceeded its credit limit in the purchase of Products, (ii) RESELLER is in default under this Agreement or any other agreement relating to RESELLER's purchase of Products, or (iii) an allocation of Products is required. No such rejection, cancellation, refusal or delay shall be deemed a breach of this Agreement by COMMODORE.

C. COMMODORE will endeavor to meet RESELLER's requested shipment dates subject to product availability and COMMODORE's production and supply schedules. However, COMMODORE shall not be liable for nonperformance or delays, which occur due to circumstances or causes beyond its reasonable control. In the event of any such excused delay or failure of performance, the date of delivery shall, at the request of COMMODORE, be deferred for a period equal to the time lost by reason of the delay. COMMODORE shall endeavor to notify RESELLER in writing of any such event or circumstance within a reasonable time after it learns of same, but shall not be liable for any failure to provide such notice. COMMODORE reserves the right to make partial shipments with the consent of the RESELLER, which consent shall not be unreasonably withheld, and invoices will be issued reflecting items shipped.

D. Any claims for defects or shortages in Products purchased by RESELLER, other than claims resulting from COMMODORE's limited consumer warranty, shall be presented in writing by RESELLER to COMMODORE within sixty (60) days of the date of shipment by COMMODORE.

E. IT IS UNDERSTOOD THAT COMMODORE MAKES NO WARRANTIES OR REPRESENTATIONS AS TO THE PRODUCTS SOLD TO RESELLER BY COMMODORE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, EXCEPT AS MAY BE SET FORTH IN COMMODORE'S LIMITED CONSUMER WARRANTY, IF ANY, ACCOMPANYING DELIVERY OF THE PRODUCTS. COMMODORE RESERVES THE RIGHT TO CHANGE THE TERMS OF SUCH LIMITED CONSUMER WARRANTY AT ANY TIME WITHOUT NOTICE AND WITHOUT LIABILITY TO RESELLER OR ANY OTHER PERSONS BY REASON OF ANY SUCH CHANGE UNLESS SUCH CHANGE IS CONSIDERED UNENFORCEABLE OR UNLAWFUL UNDER APPLICABLE LAW.

F. THE LIABILITY OF COMMODORE, IF ANY, FOR DAMAGES RELATING TO ANY ALLEGEDLY DEFECTIVE PRODUCTS, OR SHORTAGES IN SHIPMENT UNDER ANY LEGAL OR EQUITABLE THEORY SHALL BE LIMITED TO (i) DEFECTS OR SHORTAGES WITH RESPECT TO WHICH COMMODORE RECEIVES A WRITTEN NOTICE OF THE CLAIM FROM THE RESELLER WITHIN SIXTY (60) DAYS OF THE DATE OF COMMODORE'S SHIPMENT OF THE PRODUCTS; AND (ii) IN COMMODORE'S DISCRETION, COMMODORE MAY EITHER REPLACE THE PRODUCTS OR REFUND THE ACTUAL PRICE PAID TO COMMODORE BY RESELLER FOR SUCH PRODUCTS.

G. COMMODORE SHALL HAVE NO LIABILITY FOR (i) INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY KIND, or (ii) ACTUAL OR ANY OTHER DAMAGES IN EXCESS OF THE VALUE OF THIS CONTRACT.

V. PAYMENT

A. Payment for Products shall be promptly made in U.S. dollars, by the due dates set forth in the invoice, at the net invoice price, and there shall be a penalty for late payment at 11% per annum or the maximum lawful interest rate permitted under applicable law.

B. COMMODORE reserves the right at all times, either generally or with respect to any specific order for Products, to withdraw, change or limit the amount or duration of payment credit terms, if any, to the RESELLER. RESELLER further agrees not to make any deduction of any kind from any payments due to COMMODORE hereunder unless RESELLER shall have received a written credit memorandum from COMMODORE, authorizing such deduction.

C. As security for any and all indebtedness of RESELLER to COMMODORE under this Agreement, RESELLER does hereby grant to COMMODORE a purchase money security interest in favor of COMMODORE, with priority over any other security interest, in and to all of RESELLER's right, title and interest in and to: (i) all Products now owned or hereafter acquired by RESELLER; (ii) all accounts receivable and contract rights now or hereafter existing arising from RESELLER's sale, lease or transfer of Products; and (iii) the proceeds, products and accessions of and to any and all of the foregoing. RESELLER authorizes COMMODORE to perfect such security interest and will cooperate in accomplishing the same. Notwithstanding the foregoing, this security interest may be subordinated to the security interest granted to any third party, approved by COMMODORE, who provides financing to RESELLER for the purchase of Products.

VI. USE OF TRADEMARKS AND OTHER PROPRIETARY RIGHTS

A. During the term of this Agreement RESELLER may use, only in connection with and as reasonably required for, RESELLER's authorized sale, advertisement and promotion of the Products in accordance with this Agreement, the trade names, trademarks, insignias, logos, proprietary marks, and the like related to the Products and owned or controlled by COMMODORE or its affiliated companies (the "Proprietary Marks"), provided that RESELLER's use of Proprietary Marks shall be in accordance with COMMODORE's specifications, policies and directions and shall always clearly indicate that the same is the property of COMMODORE or a COMMODORE affiliated company, as the case may be, and as directed by COMMODORE. RESELLER acknowledges that COMMODORE may at any time object to a specific use or application of any of the Proprietary Marks, in which event RESELLER will cease such use or application thereof of the Proprietary Mark immediately.

B. RESELLER shall not (i) use any of the Proprietary Marks as part of any business name, except as may be approved by COMMODORE in writing; (ii) attach any name or mark to any of the Products, other than the names and marks originally appearing thereon; (iii) add to, obliterate, deface or remove any name, Proprietary Mark or serial number on the Products or packaging thereof; and (iv) make any additions, deletions or modifications to the Products without the prior written authorization of COMMODORE.

C. Nothing contained in this Agreement shall give RESELLER any other interest in the Proprietary Marks or in any patents, copyrights, trade secrets or other proprietary or confidential information related to the Products (except the non-transferable right to use to the extent required for the authorized resale of Products to Customers as herein provided), and RESELLER specifically disclaims any such interests. RESELLER agrees that its use of the Proprietary Marks shall inure to the benefit of COMMODORE and its affiliated companies.

VII. CONFIDENTIAL INFORMATION

RESELLER acknowledges that it may acquire and develop knowledge, information and materials concerning the Products, COMMODORE or its affiliated companies and their products which are and shall be the trade secrets and confidential and proprietary information of COMMODORE or its affiliated companies (hereinafter "Confidential Information"). RESELLER shall hold such Confidential Information in strict confidence and not disclose it to others, nor allow any unauthorized person, firm or corporation access to it either before or after termination of this Agreement, without the prior written consent of COMMODORE. This provision shall survive termination of this Agreement. RESELLER shall not use such Confidential Information in any way or permit others to use it in any way, commercially or otherwise, that is contrary to the best interests of COMMODORE.

VIII. RELATIONSHIP OF PARTIES

This Agreement does not in any way create the relationship of franchise, joint venture, partnership or principal and agent between COMMODORE and RESELLER. RESELLER is an independent contractor, and as such, shall not act or represent itself, directly or by implication, as agent for COMMODORE or assume or create any obligation on behalf of or in the name of COMMODORE, or otherwise bind COMMODORE in any manner.

IX. TERMINATION OF AGREEMENT

A. Either party may terminate this Agreement by written notice to the other, at any time and for any reason, upon thirty (30) days prior written notice. Either party may also terminate this Agreement effective immediately, in the event the other party breaches any term of this Agreement and such breach is not remedied within fifteen (15) days after written notice of such breach is given.

B. COMMODORE may terminate this Agreement by written notice to RESELLER, effective immediately, upon the occurrence of any of the following events: (i) if RESELLER fails to pay for the Products when payment is due or make arrangements to do so which are acceptable to COMMODORE; or (ii) upon the occurrence of any change in the financial condition or management of RESELLER which, in the sole judgment of COMMODORE, is materially adverse to RESELLER's ability to perform under this Agreement; or (iii) RESELLER defaults in any agreement with COMMODORE or with any third party providing financing to RESELLER for the purchase of Products hereunder, or (iv) if RESELLER engages in a course of conduct which, in the sole judgment of COMMODORE, substantially and adversely affects COMMODORE's reputation or its interests in the promotion, marketing or distribution of the Products, or is otherwise deemed to be just cause for termination pursuant to prevailing laws; or (v) RESELLER fails to maintain a level of sales and support for Products that is reasonably satisfactory to COMMODORE.

C. Upon termination of this Agreement, (i) all sums owing by RESELLER to COMMODORE shall immediately become due and payable without notice and COMMODORE shall thereupon have all of the rights and remedies of a seller under applicable law, and as may be provided in this Agreement, (ii) RESELLER shall cease holding itself out as a COMMODORE Authorized RESELLER and shall immediately return any Confidential Information to COMMODORE and shall remove all signs, names, insignia, logos, Proprietary Marks, and any other promotional advertising, sales, informational, technical or other material which identifies or appears to identify with COMMODORE, and deliver the same to COMMODORE, and (iii) upon COMMODORE's request, RESELLER shall assemble Products then in inventory and make them available for inspection by COMMODORE at a place and time which is reasonably convenient to COMMODORE and RESELLER. All of COMMODORE's rights and remedies shall be cumulative and no waiver of any default will affect any other or subsequent default.

D. Upon such termination, COMMODORE may elect, without obligation, to repurchase all or any part of RESELLER's inventory of Products at the lower of COMMODORE's then current RESELLER's prices or COMMODORE's original RESELLER's cost. All RESELLER Product orders received by COMMODORE and not shipped on the date that notice of termination is given or on the date this Agreement otherwise terminates shall be deemed cancelled at the option of COMMODORE.

X. MISCELLANEOUS

A. ENTIRE AGREEMENT; MODIFICATIONS; WAIVERS. This Agreement and the attached schedules set forth the entire understanding between the parties hereto and supersedes all prior understandings with respect to this subject matter, which prior understandings are hereby terminated. Such terms and conditions of this Agreement shall govern all purchases of Products by RESELLER notwithstanding any terms and conditions set forth in RESELLER's purchase order. No modification or alteration shall be binding unless executed in writing by the parties. No waiver of any provision of this Agreement (whether or not similar) shall be construed a continuing waiver unless expressly so stated.

B. GOVERNING LAW AND JURISDICTION. This Agreement shall be governed by and construed in accordance with the laws of the Commonwealth of Pennsylvania. The parties agree to submit to the exclusive jurisdiction and venue of the appropriate courts located in Pennsylvania for the purpose of any suit, action or other proceeding in connection with this Agreement. The parties expressly waive any and all objections to jurisdiction or venue in any of such courts and hereby consent that service of process in any litigation may be served in the same manner as any notice hereunder as set forth below.

C. UNENFORCEABILITY. If any part of this Agreement is declared invalid or unenforceable by a government authority or court of competent jurisdiction from which decision no appeal is or can be taken, the remainder of this Agreement shall remain valid, unless the removal of such invalid or unenforceable part shall, in the opinion of COMMODORE, have the effect of materially nullifying or impairing this Agreement.

D. AUDITS. COMMODORE reserves the right to audit and inspect RESELLER's books, records and operations reasonably necessary to verify proper compliance with this Agreement. COMMODORE shall give at least ten (10) days prior written notice of such audit and conduct such audit at its own expense and solely for the purpose of insuring compliance with this Agreement. Any such audit or inspection shall occur during regular business hours and shall not unreasonably interfere with RESELLER's business activities. All information obtained shall be considered confidential.

E. NOTICE. Any notice required or permitted to be given under this Agreement shall be in writing and either delivered personally or sent by telex, telegram or deposited in the mail, postage prepaid, registered or certified, return receipt requested, addressed to the parties at the address appearing on the last page of this Agreement, and shall be deemed given three (3) days after the date of mailing or on the date of personal, telex or telegram delivery.

F. ASSIGNMENT. This Agreement and the rights and benefits hereunder shall inure to the benefit of the parties and their respective successors and assigns except that, due to the personal nature of RESELLER's commitments hereunder, neither this Agreement nor any rights or duties hereunder are transferable or assignable or delegable by RESELLER, either voluntarily or by operation of law.

G. EXPORT RESTRICTION. RESELLER agrees that it does not intend and will not ship or transmit (directly or indirectly) any Products purchased under this Agreement or any technical information furnished by COMMODORE to RESELLER with respect to the Products, in violation or contravention of any rule or regulation relating to exports of the U.S. government or any member nation of COCOM which has jurisdiction over COMMODORE, its parent, affiliated and subsidiary companies.

IN WITNESS WHEREOF, the parties hereto have executed this agreement as of October 11, 1988.

RESELLER: _____

By: _____

Title: _____

Signature: _____

Address: _____

RJG/sgr/1

COMMODORE BUSINESS MACHINES, INC.

By: _____

Title: _____

Signature: _____

Address: 1200 Wilson Drive

West Chester, PA 19380

VAR BULLETIN

DATE: March 22, 1990

SUBJECT: General Policies

In order to insure the end-user satisfaction that is critical to the success of the Amiga and Professional Series products lines ("Products"), it is essential to provide such end-users, who typically have little familiarity with such technically complex products, with high quality individualized presale and postsale support. It is also essential to maintain trained personnel, an inventory of hardware and software products required for the "Solution-Sell", and a facility reasonably sufficient to ascertain and supply the end-user's requirements. In addition, Commodore has established a Manufacturer's Program and Records Retention policy to further ensure that the above programs are implemented as intended and continue to benefit the end-user, you as a Commodore VAR and Commodore.

SALES POLICY:

VARS are business organizations that resell Product to authorized end-user customers with significant Value added to the Product prior to such sale. Value must be added through specific proprietary application software, proprietary hardware or proprietary peripherals, which are appropriate to the VAR solution as described to Commodore.

VARS are required to make every reasonable attempt to understand the requirements of the end-user customer so they can recommend a solution specific application that may solve the customers' business requirement. Furthermore, the VAR is required to make every reasonable effort to support the end-user customer after the delivery of the Product and the integrated VAR Solution.

All sales of Products must be based on a face to face meeting with the business customer and VAR. This meeting can occur at a VAR business location that has been approved by Commodore, an approved trade show or similar event or at the end-user customer's site by employee of VAR.

Commodore prohibits the sale of Products by means of the methods of catalog or mail-order delivery as well as the sale of Products to any entity for which the VAR is not authorized. In the case of VAR Resellers this would expressly prohibit the sale of Products independent of the significant proprietary software, hardware, and/or peripherals which are to be added by Reseller.

All sales of Product by VAR are intended for use in the United States of America and must not be knowingly sold for uses external to the United States without the express written permission of Commodore.

TRAINING POLICY:

The VAR location must be capable of performing comprehensive, hand-on demonstrations of the VAR Solution which they sell. The VAR location must be uniquely knowledgeable with respect to specific market expertise and the resulting VAR Solution and must be able to train the business end-user in the Product and in the VAR Solution.

In order to best represent the Product and therefore service the end-user customer the VAR must maintain a staff of qualified sales, service, and technical personnel meeting the standards set by Commodore. This staff must be willing to respond to questions concerning the support of the Products and VAR Solution.

MANUFACTURER'S PROGRAM POLICY:

Commodore will from time-to-time offer promotional programs to support the marketing of Product through the VAR base. Such programs shall be governed solely by written bulletins from Commodore's headquarters describing the same. VARS are expected to participate in these programs to encourage the development of potential solution markets.

RECORDS RETENTION POLICY:

The VAR must maintain sales records (ie. invoices) for all products sold. These records must specifically list serial numbers for the Products sold and include the end-user customer name, company, address, telephone number and date of sale.

These records must be maintained for a period of three years from the date of sale. These records must be made available to Commodore upon all reasonable requests, to inspect compliance with Commodore's policies and procedures.

VAR SCHEDULE

These schedules relate to the contract dated _____ by and between COMMODORE and the related VAR identified below. Due to the information set forth in these schedules they are subject, from time to time, to ammendment by COMMODORE, in its sole discretion.

SCHEDULE A Products:

(Will only include those required for the VAR solution. Must be specific) Reseller is not authorized to sell Products independent of such VAR Solution.

SCHEDULE B Customers:

Retail end-users, the significant portion whose use of the Amiga is devoted to the VAR Solution defined as follows: _____. The VAR Solution is for the end-users productive use and is not for resale, rent or lease. End-users whose use of the VAR Solution will not be significant are excluded from this definition.

SCHEDULE C Location(s):

Unless otherwise indicated herein, the single approved location that is identified below adjacent VAR's authorized signature. This location can be an office or a retail storefront providing it is commercially appropriate for the VAR Solution.

SCHEDULE D Policies and Programs:

This schedule incorporates by reference COMMODORE's VAR Operations Guide and such VAR bullentins as have been distributed to COMMODORE VARS and are currently in effect (copies attached).

Effective Date: March 23, 1990 This document supercedes such schedules as have been previously issued relating to the subject matter hereof.

RESELLER: _____ COMMODORE BUSINESS MACHINES, INC.

By: _____ By: _____

Title: _____ Title: _____

Signature: _____ Signature: _____

Address: _____ Address: 1200 Wilson Drive

_____ West Chester, PA 19380

**Commodore Business Machines
OEM Sales Division**

I hope you find the enclosed material both helpful and complete in meeting your needs. You should find information useful for integrating our products into your line of products.

Contents:

- * OEM Relationship Overview
 - * Product Literature
 - * Developer Program Application
 - * Service Center Program Application
 - * Credit Application
 - * OEM Agreement (Sample)
-
- o We need to process at least the Credit Application to release an order for development, testing or ongoing order purposes. Send these completed documents directly to my attention.
 - o Standard Terms are net 30 days on approved credit.
 - o All orders must meet a minimum of \$2,500 per shipment.
 - o All orders are shipped FOB Commodore warehouse Pennsylvania, California or Hong Kong collect.

For more information please contact me at:

(215) 431-9481 (Telephone)
(215) 431-9464 (Fax)

Regards,



Donna O'Neil
Technical Support
Liaison, OEM/VAR Markets

This Agreement, made and entered into as of _____, 19____, by and between Commodore Business Machines, Inc., having a place of business at 1200 Wilson Drive, West Chester, Pennsylvania (hereinafter referred to as "CI") and _____, having a place of business at _____ (hereinafter referred to as "Company").

In consideration of the mutual covenants and conditions herein set forth, the parties hereto do agree as follows:

1. DEFINITIONS

For purposes of this Agreement, the following terms shall have the following meanings:

A. "Assembly" shall mean the items sold to Company by CI, which items are more particularly in attached Exhibit A.

B. "Licensed Rights" shall mean technical know-how and specifications relating to the Assembly and necessary for the performance of the rights and obligations set forth herein, which are owned or controlled by CI during the term of this Agreement.

C. "Company Products" shall mean the products manufactured and marketed by Company and described in attached Exhibit B, which will include Assemblies as an integral and essential component as well as substantial software and/or hardware additions developed or manufactured by Company specifically for incorporation into and essential components of such products.

2. PURCHASE OF PRODUCT

A. During the term of this Agreement, CI shall sell to Company, and Company shall purchase from CI, Assemblies for use as components in Company Products according to the Schedule set forth in attached Exhibit C. In addition to Assemblies, CI shall sell to Company sufficient components for such Assemblies to permit Company to stock repair parts to provide customer support for Company Products.

B. Company will provide to CI periodic order forecasts, and other information relating to market conditions as may from time to time be reasonably requested by CI. Company will provide to CI at least ninety (90) days prior written notice of any changes to the order forecast.

C. CI does not represent, warrant or guarantee that the Assembly:

- (i) is or will be compatible or operable with, or as a component or part of, Company Products; or
- (ii) will remain identical to the Assemblies being manufactured and sold as of the date of this Agreement; or
- (iii) as incorporated into Company Products will comply with any requirements of the Federal Communications Commission, which compliance shall be Company's responsibility.

D. The sale by CI and purchase by Company of Assemblies shall be subject to the provisions of this Agreement including the terms and conditions described in attached Exhibit D.

E. Company shall place its first order so that delivery will be scheduled no later than _____.

3. LICENSE

A.1. CI hereby grants to Company with each Assembly sold to Company hereunder a royalty free, non-exclusive right under the Licensed Rights, to use the Assembly solely as a component in Company Products, and for no other purpose of any kind or nature.

2. Except as specifically provided above, all other rights under the Licensed Rights are reserved by CI. Company shall have no right or license, directly or indirectly, to manufacture or have manufactured, produce or duplicate the Assembly or sell, transfer, license, lease, market or distribute the Assembly, without such substantial additions as are set forth in the definition of Company Product, or as a component of or addition to any other product.

B. CI reserves the right to audit and inspect Company's books, records and operations to verify the proper use of the Licensed Rights. CI shall conduct such audit at its own expense and solely for the purpose of insuring compliance with this Agreement. Any such audit or inspection shall occur during regular business hours and shall not unreasonably interfere with Company's business activities. CI shall give Company fifteen (15) days prior written notice of the date of such audit or inspection and the name of the firm or individual who will be conducting the audit or inspection. All information obtained shall be considered confidential.

4. TERM

This Agreement shall commence effective as of the date hereof and shall continue for a three (3) year period thereafter.

5. TECHNICAL SUPPORT

CI shall furnish to Company the technical information and technical assistance described in attached Exhibit E.

6. EXPORT RESTRICTION

Company agrees that it does not intend and will not ship or transmit (directly or indirectly) any Assembly purchased under this Agreement or any information embodied in the Licensed Rights or any other technical information furnished by CI to Company with respect to the Assembly, in violation or contravention of any rule or regulation relating to exports of the U.S. government or any member nation of COCOM which has jurisdiction over CI, its parent, affiliated and subsidiary companies.

7. ADVERTISING

A. Company agrees that it shall provide the following credit, or its substantial equivalent, to appear on the initial visual display on Company Products which use the Assemblies: "Produced with the Commodore AMIGA microcomputer".

B. Company also agrees to allow CI to use Company's name and the name or designation given by Company to the Company Products, in CI's advertising and promotional materials.

C. Company shall refrain from removing or otherwise defacing any CI copyright notices, trademarks and logos existing on such Assemblies.

8. CONFIDENTIAL INFORMATION

A. All information disclosed by either party under this Agreement which is identified in writing by the disclosing party as confidential (hereinafter collectively referred to as "Confidential Information"), shall be held by receiving party in the strictest confidence and shall not be disclosed to any third party or otherwise used, published or made public, except as may be reasonably required in the exercise of the rights and licenses granted under this Agreement, provided, however, that information shall not be deemed to be Confidential Information if it is (i) in the public domain through no wrongful act of the receiving party; (ii) rightfully received from a third party without restriction and without breach of this Agreement, and which third party is itself not bound by a restriction of non-disclosure; (iii) approved for release by written authorization of the disclosing

party; or (iv) already within the recipient's possession or knowledge, or which is independently developed and which possession, knowledge, or independent development shall be substantiated by documentary proof. The receiving party warrants and represents that it will take all necessary, reasonable measures to protect the Confidential Information disclosed pursuant to this Agreement. This Section 9 shall survive termination of this Agreement.

B. The receiving party further agrees that it shall not make any disclosure of any or all of such Confidential Information to anyone, except to its employees and contractors to whom such disclosure is necessary to the use for which the rights are granted hereunder. The parties shall appropriately notify each employee and contractor to whom any such disclosure is made that such disclosure is made in confidence and shall be kept in confidence by such employee or contractor.

9. TERMINATION

A. This Agreement may be terminated if either party materially defaults in performance of any of its obligations under this Agreement, whereupon the aggrieved party shall notify the other party in writing of the alleged default. Such written notice shall specify the default and state the aggrieved party's intention to terminate if the default is not cured, whereupon, except for breach of Paragraph 2.E., the recipient of the notice shall have thirty (30) days to cure the default and, upon a failure to so cure, this Agreement shall automatically terminate.

Notwithstanding the foregoing, it is acknowledged by the Parties that any breach or threatened breach of the provisions of Paragraphs 3 and 9 of this Agreement shall cause immediate and irreparable harm and substantial injury to CI and accordingly CI shall be entitled to immediate injunctive relief in any court of competent jurisdiction which award may be entered in any court of competent jurisdiction.

B. Upon termination of this Agreement, all Licensed Rights shall revert to CI and Company shall cease the exercise of the Licensed Rights except that CI at its option may elect to (i) purchase Company's inventory of Assemblies that have not been incorporated into Company Products at a reasonable price no greater than CI's original sale price or (ii) permit Company to dispose of any Company Product inventory, in existence at the time of termination in accordance with the terms of this Agreement.

10. MISCELLANEOUS

A. All notices, payments, and statements in connection herewith shall be in writing and shall be addressed as follows:

TO:

TO: Commodore Business Machines, Inc.
1200 Wilson Drive
West Chester, Pennsylvania 19380
Attention: Vice President Sales

or at such other address as the respective parties may designate in writing subsequent to the signing of this Agreement. Such notices, payments, and statements shall be rendered by depositing them, addressed as aforesaid, certified or registered mail, postage prepaid, in the mail and shall be deemed given on the date on which they are mailed.

B. This Agreement shall be governed by the laws of the Commonwealth of Pennsylvania, except as to bankruptcy, patents, trademarks, and copyrights which are governed by federal law of the United States.

C. This Agreement shall be binding upon and inure to the benefit of the heirs, successors and assigns of CI and CI's direct or indirect, parent, affiliated and subsidiary companies and Company and Company's direct and indirect parent, affiliated and subsidiary companies and their heirs, successors and assigns, provided, however, that neither party shall assign any of its rights or delegate any of its obligation under this Agreement to any third party except as is expressly set forth herein. CI may assign its rights and delegate its obligations hereunder i) pursuant to a merger, a sale of all of the stock of CI or the sale of all or substantially all of its assets, which are related to the business which is the subject of this Agreement or ii) to any affiliate of CI, to wit: Commodore International Limited, a Bahamian corporation ("CIL"), Commodore Electronics Limited, a Bahamian corporation ("CEL"), or any other company in which more than fifty percent (50%) of the stock entitled to vote for the election of directors is now or will be hereafter owned by CI, CIL or CEL, either directly or indirectly. Upon any such permitted assignment, the assignee shall be bound to all of the obligations of the assignor.

D. The only relationships between Company and CI that are intended to be created by this Agreement are those of supplier and purchaser and licensor and licensee. Neither party shall be, nor represent itself to be, an agent, employee, partner or joint

venturer of the other, nor shall either party transact any business in the name of the other, or on the other's behalf, or in any manner or form make promises, representation or warranties or incur any liability, direct or indirect, contingent or fixed, for or on behalf of the other party.

E. This Agreement and the Exhibits attached hereto constitute the entire Agreement between the parties hereto with respect to the subject matter contained herein and supersedes any previous oral or written communications, representations, understandings or agreements. All representations under this Agreement shall survive termination of this Agreement.

F. This Agreement may be amended, modified, superseded or cancelled, and any of the terms herein may be waived, only by a written instrument executed by each party hereto or, in the case of waiver, by the party or parties waiving compliance. The delay or failure of any party at any time or times require performance of any requirement hereof shall in no way affect its rights at a later time to enforce the same. No waiver by any party of any condition or of the breach of any term contained in this Agreement, whether by conduct or otherwise, in any one or more instances, shall be deemed to be or construed as a further or continuing waiver of any such breach or of the breach of any other term of this Agreement.

IN WITNESS WHEREOF, as of the date first written above, the parties have caused this Agreement to be signed by its duly authorized officers.

COMPANY

COMMODORE BUSINESS MACHINES, INC.
1200 Wilson Drive
West Chester, Pennsylvania 10380

BY: _____

Typed Name: _____

Title: _____

BY: _____

Typed Name: _____

Title: _____

FDM/2103:pdh

EXHIBIT A
PRODUCTS

EXHIBIT B

PRODUCTS

EXHIBIT C
ASSEMBLY DELIVERY SCHEDULE

<u>Date</u>	<u>Quantity</u>	<u>Price*</u>
-------------	-----------------	---------------

* Such prices as are set forth herein are based upon current costs and purchases in accordance with the quantities specified. CI reserves the right to pass on to the Company actual cost increases which are incurred by CI for such Assemblies ordered by Company after the first twelve months under this Agreement. CI also agrees to pass on to Company, on an equal percentage basis, any reduction in the published dealer price (other than reductions resulting from temporary promotional programs) which is applicable at the time of acceptance of any purchase orders submitted hereunder. In the event Company fails to comply with the quantity commitment set forth herein, CI reserves the right, in addition to any other legal remedy it might have, to impose a fifteen (15%) surcharge on all orders received during any period on which such quantity commitment has not been met.

EXHIBIT D

TERMS AND CONDITIONS

A. All orders for Assemblies shall be minimum lots of 300 and delivery shall be FOB _____. The price per unit for Assemblies shall not include any transportation, insurance, duty or taxes, of any nature. All such amounts shall be paid by the Company.

B. Title to all Assemblies and all risks of loss or damage in transit or otherwise, shall pass to the Company when delivered to the carrier or, if the Company has requested a delay in shipment, when placed at the Company's disposal as determined by CI, at CI's place of distribution.

C. All orders submitted for Assemblies, shall be firm, noncancellable orders which shall be binding upon the CI upon its acceptance. Unless otherwise agreed to in a writing signed by both parties, the Agreement and these Terms and Conditions shall govern all sales and purchases of Assemblies, notwithstanding any terms and conditions as set forth in the purchase orders of either party to this Agreement.

D. The delay or inability to deliver Assemblies resulting from causes beyond the reasonable control of CI, or the failure to give notice of the same shall create no liability. The delivery date may be extended for a period of time equal to such delay.

E. All Assemblies shall perform in accordance with technical specifications and be warranted free from defects in materials and workmanship for a period of ninety (90) days from receipt by Company. In the event Company discovers such defects, it shall return the defective part of the Assembly to CI who shall, at CI's sole option, repair or replace such defective part.

F. EXCEPT FOR THE ABOVE, THERE ARE NO WARRANTIES OR REPRESENTATIONS, INCLUDING WITHOUT LIMITATION, IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS OF A PARTICULAR PURPOSE. IN NO EVENT SHALL CI BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY KIND, NOR TO ANY INDIVIDUAL OR ENTITY OTHER THAN COMPANY NOR FOR DAMAGES IN EXCESS OF VALUE OF THIS CONTRACT.

G. Payment for the Assemblies shall be made within thirty (30) days of receipt of such Assemblies by Company at the net invoice price by letter of credit for foreign delivery and in U.S. dollars for domestic delivery. Late payments shall accrue interest

at the maximum lawful rate. CI retains a purchase money security interest in all Assemblies shipped until the entire purchase price has been paid to and received by CI. Company authorizes CI to perfect such security interest and will cooperate in accomplishing the same.

H. CI reserves the right at all times, either generally or with respect to any specific order for Assemblies, to withdraw, change or limit the amount or duration of payment credit terms, if any, to the Company. Company further agrees not to make any deduction of any kind from any payments due to CI hereunder unless Company shall have received official written credit memorandum signed by CI authorizing such deduction.

I. Nothing contained in the Agreement shall give Company interest in, or right to use the trademarks "Commodore" or "Amiga" or other trade names, trademarks, insignias, logos, proprietary marks, and the like, owned or controlled by the CI or its affiliated companies, or any patents, copyrights, trade secrets or other proprietary or confidential information related to the Assemblies, except the use thereof specifically and expressly provided in the Agreement, and Company specifically disclaims any right in any such Intellectual Property Rights.

COMPANY

CI

By: _____

By: _____

EXHIBIT E
TECHNICAL SUPPORT

Technical Support to be furnished by CI will be the standard support package available via the CI Independent Developer Support Group.

Additional technical support may be requested by Company and CI will provide the same, at the then prevailing rates, and to the extent the required resources are reasonably available.

1

2

3



Scalable Fonts: A Decade of Change

By Bob Burns

The preparation of text documents on personal computers has come full circle. In the late 1970s, the state of the art in presentation graphics used plotters and characters formed from arcs and lines to produce output. The characters could be put through the same graphic transform matrix as the other elements of the vector picture and could thus be scaled to the desired size. Users were attracted by the quick turnaround achieved with the use of the personal computer and were willing to compromise on the typefaces available to render their text. The most popular fonts were those developed by Dr. Hershey for the U.S. Government which were distributed through the U.S. National Bureau of Standards. These vector fonts produced good plotted output and were available in three styles: Roman, Sans-serif, and Fraktur. The Roman and Sans-serif fonts were available in normal and bold weights by doubling and tripling the pen strokes. The Hershey fonts have distinctive characteristics: for example, the hook on the arm of the capital R. I remain amazed at the contribution of Hershey fonts to the industry -- I still see Hershey fonts in printer font packages advertised today. But printers in the late 1970s were primarily daisy wheel printers. The graphics printers that did exist had poor resolution. The struggle for these 9 pin graphic printers was to create a legible letter form that was also differentiable from its italic or bold version. The resolution to create letter forms that were recognizable as Times Roman vs. Garamond was not yet available.

In the early 1980s, the laser printer and its 300 dpi resolution allowed bitmap characters to match the detail of the 1000 dpi plotter with "8 dot wide" pen. Hewlett Packard's LaserJet led the way with fonts that were reminiscent of the classic Times Roman and Helvetica typefaces, both by their form and by their abbreviated names TmsRmn and Helv. The resolution to distinguish Times Roman from Garamond was now available. But scaling a bitmap of a character produced unacceptable results. Each character of each typeface had to be hand-edited for each display size. The production of these bit-mapped fonts was thus labor-intensive and expensive. This work was not performed by the traditional print type houses but by the printer manufacturers and digital type houses instead.

In 1984, Adobe's PostScript[™] broke new ground and went back to the character outlines to generate the bitmap characters for the laser printer. Using proprietary technology called *hinting*, they arranged to provide the brand name typeface outlines of the traditional print

type houses. The *hinting* eliminated the need to hand edit the fonts -- it provided good quality output at arbitrary sizes. These outlines are composed of arcs and lines like the earlier days, though the arcs are typically Bezier instead of just circular, and the path describes the outline of the font instead of the pen path. These outlines can have standard graphical transforms applied to them. *Font hints* may then be used to fit the resulting scaled outline to the final pixel grid and correct any differences in roundoff so that the result is pleasing. For example, stem widths are made consistent. This is the core of what scalable fonts are: outlines that can be scaled to arbitrary sizes and look good. This gave the type houses entry into the fast growing desktop publishing market without massive effort and gave the users familiar fonts to use. The proprietary nature of Adobe's *hinting* technology gave them a lock on the market -- but only for a few years.

The technology for generating characters continues to progress. Adobe's grip on the industry is loosening as competing solutions appear. The industry of fonts on computers falls into three camps: font suppliers, font engines, and font users. All suffer from the conflicting desire to support an industry standard for electronic type yet maintain the leading role in the industry.

Font suppliers, specifically the major print type houses like Linotype, Monotype, ITC, and Compugraphic, now appear to realize the importance of electronic type as a source of revenue for their business. When Adobe first introduced PostScript™, only Linotype quickly arranged for their outlines to be used. Now press releases for announced font engines routinely mention several type houses that have committed to convert their outlines for use with that engine. That these major suppliers are now competing to provide their outlines is good for everyone, except perhaps some of the young digital type houses.

Font engines are now available from several sources. Adobe's success affected the industry in many ways. First, users want the benefits of scalable fonts for their non-PostScript™ printers -- primarily for the HP LaserJet and its clones connected to PCs. Bitstream's Fontware and Agfa Compugraphic's Typographer are two examples of user products to fill that need. Second, computer operating system suppliers are frustrated by the fact that the market's focus has not been on the computers, but on the print engine connected to them. Both Apple (Macintosh) and Microsoft (MS-DOS) feel like the tail is wagging the dog. The assimilation of PostScript™ into the OS environment itself using Display PostScript™ for the complete imaging model, as NeXT™ has done, is one way to resolve the conflict -- but it keeps Adobe in the relationship. Computer manufacturers want control of the software environment so that they can differentiate themselves from one another as they jockey for market pre-eminence.

Font houses are frustrated by their inability to compete on a level playing surface. Adobe controls PostScript™ hinting technology, so font vendors must either strike a deal with Adobe or release unhinted fonts that are of poorer quality. Adobe itself is now producing fonts, creating an apparent conflict of interest in facilitating the conversion of third party fonts to hinted form.

Adobe reacted to these pressures by announcing that it will open up its hinting technology, but that decision is too little too late. Their offer will only address the concerns of those font houses so frustrated by Adobe's control of the font conversion process that they are willing to pay Adobe a licensing fee. And competition to Adobe's technology is already here.

Apple has announced that it will split from reliance on Adobe, build on Quickdraw as an imaging model and will release a new font engine called Royal. HP has put its weight behind Compugraphic's Intellifont font engine, used by the Typographer application for the PC and by Gold Disk's applications for the Amiga. Intellifont will be an integral part of HP's next generation printer control language, but it appears that HP's attempt to make Intellifont the engine for OS/2 Presentation Manager (PM) has failed. HP and Imagen's attempt to challenge PostScript™ with the more robust DDL failed when Imagen was unable to deliver it, but some of Imagen's principal staff formed Folio, recently acquired by SUN, which is offering OpenFonts. OpenFonts consists of the *F3* font format, the *TypeMaker* tool to automatically convert other font formats to the *F3* format, and *TypeScaler* which is the font engine that produces bitmaps from the *F3* outlines.

Most surprisingly, Microsoft has announced that it will be using Apple's Royal technology. Microsoft has hedged the commitment by also announcing it will provide hooks to support any font engine in its imaging model. But if this Apple/Microsoft commitment remains intact, Royal will be the strongest contender for dominance in the font engine arena. Moreover, indications from Microsoft are that the Royal font engine will be available to third parties for modest fees. I think Apple and Microsoft have agreed to eliminate the driving role of the font engine in the electronic publishing solution by allowing Royal to become an inexpensive commodity, each hoping to regain market control with solutions focused around their respective proprietary operating systems and imaging models. This shakeup can be good for Commodore, too. The Amiga has the strength of experience in many of the aspects of a graphical multi-tasking environment on personal computers.

The Amiga Challenge

The Amiga of the 1980s is missing only a few elements in the challenge for superiority. One of these is the weakness of its imaging model caused by its great flexibility. That can be compensated for by the programmer. The more difficult failing is the lack of robust font support. It is just impossible to produce good text output on both the varying display resolutions and the different printer resolutions using the current bitmap text model. This is why Commodore has committed to scalable font support for the 1990s.

The industry today is treating font engines like razors, and the fonts available for them like the razor blades. The engines are promoted heavily and are licensed for what is next to nothing on the corporate level. The profit is in the sale of fonts to the end user. For the Amiga, we must be sensitive to the fact that the user makes purchases based on value, and work to make the fonts available at the lowest possible price. With the font industry as unsettled as it is, we will achieve this by providing a promising solution as soon as possible, structured so that applications will be insulated from the specifics of the font engine vendor, should it be appropriate to change later.

Scalable Fonts on the Amiga

Commodore has acquired Agfa Compugraphic's Intellifont code. Intellifont has several advantages that make it appropriate for our use. It is a mature and proven engine: we have it today and can see its results. It also has the recommendation of Hewlett Packard. Fonts are available for it today, both on 3.5" MS-DOS disks through several channels and on AmigaDOS disks through Gold Disk at more reasonable prices.

Integration of the Agfa Compugraphic Intellifont font engine into the Amiga system software will be performed in three phases, the first of which will provide a way for Amiga applications to use fonts created by Intellifont using the Amiga text-related system calls that already exist. Specifically, the Intellifont font engine will be used to create *diskfont.library* compatible Amiga fonts. This approach gives a fast start to basic Intellifont access -- Intellifont will be used to generate arbitrary font sizes from the outline data, including limited control of features like character slant and emboldening.

A



Compatibility: *How to Make Sure Your Programs Will Work in V2.0 and Later Versions of the Amiga OS*

by Mike Sinz

The Amiga hardware and operating system software were designed with flexibility in mind. This has allowed newer models of the Amiga to incorporate advances in technology such as faster processors and higher capacity RAM chips without sacrificing software compatibility. The newest release of the Amiga operating system, Amiga OS V2.0, includes major enhancements to the system software to go along with the many improvements in hardware that have been happening over the past few years. These changes will help make the Amiga a more competitive product, however, they also present a problem; inevitably some of the changes in the system software will have undesirable side-effects on existing applications.

These side-effects can range from mere cosmetic problems caused by new default colors in Preferences to serious flaws (or worse) caused by new display capabilities. Commodore has made every effort to maintain compatibility with currently available software and for most applications, the changes are invisible. This article outlines the major issues in compatibility and the points you should look for when testing your software and hardware products with V2.0 or the Amiga 3000.

New Features and Configurations

Many of the compatibility issues are visual and stem from the new interface look and the fact that the system is more configurable. For instance, there is now more complete support in Intuition for overscan screen sizes. It is also easier for the user to select the various Workbench screen resolutions, the font to use, and even whether to use a PAL or NTSC display. Most all of the system defaults can now be easily changed by the user.

Compatibility problems also arise from the use of system structures which are undocumented or listed as "private". One of the more common of these is the use of private areas in IntuitionBase to try to "force" certain features or options (such as overscan screens with full mouse travel). Many system structures have changed. Applications which relied on these structures will have problems.

Another compatibility issue - one which is often overlooked - is the fact that the Amiga 3000 does not have a 68000 processor, it has only the 68030. This means applications which rely on the characteristics of the 68000 (such as its speed) will not operate correctly on the Amiga 3000. You need to worry about this only if you ignored Commodore's developer guidelines.

Larger Address Space

A related issue is that both V2.0 and the Amiga 3000 directly support memory in the address range above the 24-bit addressing limit of the 68000. The operating system has always worked with memory above the 24-bit addressing limit but complete support for such memory was missing until now. With the Zorro III 32-bit system bus and the Amiga 3000 motherboard, even a base Amiga 3000 has memory that is above the 24-bit address limit. This means that you will now start getting memory addresses and pointers that have something other than zero in the upper byte.

This will be a problem for anyone who ignores Commodore developer guidelines by using the upper byte for flags or other purposes. A more subtle possibility for a problem is a loop that goes one byte too far. This happens most with strings and string buffers that are on the stack. For instance, suppose you have stored a string on the stack and after the string is some pointer or perhaps the return address of the routine. In a system where memory is limited to 24-bit addresses, a copy operation which goes one byte too far will not be noticed because the upper byte of the address following the buffer is zero anyway. However, with addresses beyond the 24-bit range the upper byte of an address on the stack is not zero and a copy operation which goes one byte too far will change that address. Under these circumstances, the bug never appears until the software is run on systems which use the full 32-bit address range.

Hardware Compatibility

For hardware designers, it is important to make sure that the physical dimensions and external connectors of plug-in boards are within Commodore specifications (see the A500/A2000 Technical Reference Manual). With the A2000, there was some extra room inside the machine for expansion boards but this is now gone on the A3000. For example, there are video slot boards which fit in the A2000 that do not fit in the A3000.

The AUTOCONFIG process has also changed. You can no longer assume that two adjacent PICs will be configured in two successive AUTOCONFIG memory areas. Since the new AUTOCONFIG routine first tries to find out what expansion boards are out there, it could decide that a different arrangement is more efficient.

There are many other points to consider when testing your product for compatibility with V2.0 and the Amiga 3000. These are summarized in the outline below. Use the outline as a checklist in your testing efforts.

V2.0 Compatibility Checklist

I. General changes and issues

1. Direct ROM calls will never be supported.
2. Depending on CPU or memory timing to make the program pace itself will not work. If you need to do critical system timing, use the `timer.device` or `cia.resource`. The CPU and memory sub-systems are not designed for timing.
3. Depending on the fact that `ExecBase` or some other system structure is in a specific location will not work. For example: `ExecBase` in many machines happens to be at `$0676`, however, this is not guaranteed. On some machines `ExecBase` may be elsewhere.
4. Depending on the system to have a certain amount of memory free will not work. Various parts of the system may need more or less memory than older versions and various hardware configurations can change this.
5. Counting on the system to have ONLY `MEMF_CHIP` memory or at least some `MEMF_FAST` memory will not work. Likewise, the fact that the system does not have any `MEMF_FAST` memory does not guarantee that it has less than 1 meg (or 2 meg).
6. Depending on machine hardware states when taking over the machine can be very dangerous. For example, in V2.0 CIA-A, timer A has the interrupts turned on by default.
7. Counting on a certain amount of system stack usage is dangerous. Various functions in the system may change in the amount of stack used. Also, any call that results in a call to `AllocMem()` can end up using large amounts of stack since `AllocMem()` may end up needing to go through an `Expunge()` process that calls the `Expunge()` vectors of various libraries and devices. The `Expunge()` functions cannot allocate their own stack (they may have been called from `AllocMem()`) hence they use the stack of the calling function. This kind of problem can be very difficult to find since it only occurs during a memory panic on a system with lots of libraries which the user is no longer using.

8. Depending on undocumented side-effects of system functions is dangerous. When the functions change so will the undocumented side effects.
9. Setting flag bits that are undefined or using private, reserved, or undefined fields in systems structures can cause you to break.
10. Undefined bits in hardware registers should be set to zero as they may, in the future, be used for new modes. For instance, the ECS chips have some new bits in various registers. It is important that they are not set to some random value. The new registers are designed to do nothing when set to zero.
11. Do not write to read-only registers or to addresses in IO space that do not currently have registers.
12. Do not read from write-only registers or from addresses that do not currently have registers.
13. After a RESET instruction from the CPU, all AUTOCONFIG devices are de-configured. This means that AUTOCONFIG memory would no longer be there. If you need access to items on an AUTOCONFIG device, you cannot RESET.

II. Changes to exec.library

1. AddTask() works differently under V2.0. If a program manipulates the task list, it will now break.
2. AddTask() now needs to do a small AllocMem() in order to implement the new features of Exec. Because of this, AddTask() may now fail in low-memory conditions.
3. New, two-pass AUTOCONFIG, configures all RAM boards before other devices.
4. ExecBase/SysBase is now in MEMF_FAST memory, if available. This means that after a RESET instruction, ExecBase may no longer be valid. This may cause problems for some copy-protection methods that have custom boot blocks.

5. Supervisor stack is now in MEMF_FAST memory, if available.
This means that after a RESET instruction, the SSP may no longer be valid. Also no supervisor stack operations may be done after a RESET. This may cause problems for some copy-protection methods that have custom boot blocks.
6. Exception vectors may now be in MEMF_FAST memory, moved there via the VBR in 68010 and higher CPUs. Direct hacking at the exception vectors WILL break. You should use Exec to modify vectors.

III. Changes to layers.library

1. SIMPLE_REFRESH layers are a bit smarter under V2.0.
Moving a simple refresh layer that was not obscured will no longer cause a refresh event for that layer. Unfortunately, if your program relied on the refresh event to tell when the window moved, this will no longer work.
2. Sizing a SIMPLE_REFRESH layer does not clear it. An old side-effect to sizing a simple refresh layer was that the whole layer was cleared and a refresh event for the whole layer occurred (since it was cleared). The new system does not clear the layer; any part of the layer that existed before the size operation occurred is not given a refresh event. Since Intuition windows are built on these layers, the same effect is seen in Intuition windows. Because of this, if your program relies on a refresh event after a sizing operation to refresh the window, it will not work correctly. If the contents of the layer need refreshing after sizing, you will have to clear and refresh those areas.
3. Layers now runs at different speeds. Some operations that used to be slow may now be much faster. Under certain conditions, some operations may take longer (they now do better de-dicing). It is very important that you NOT rely on the timing of the layers, other than that they will be as fast as possible. Using the fact that moving a layer took some fixed amount of time in order to pace an animation or other activity will never work correctly as not only can the layers.library change, so can the CPU, which is a major part of layers speed.

6. Use the `NewLayerInfo()` function to create the layer structures. Do not use `FattenLayerInfo()`, `ThinLayerInfo()`, `InitLayers()`. (They are now obsolete.)
7. Some parts of the layers structure has change. While for 2.0 this has been limited to the areas that were reserved for future changes. However, in order to try to stay compatible with layers beyond 2.0, developers should not directly manipulate any elements of the layer structure or, if there is no way around it, should limit themselves to the `*front`, `*back`, `*rp`, `bounds`, `Flags`, `*SuperBitMap`, and `*DamageList` elements of the structure.

IV. Changes to graphics.library

1. Any poking into private areas of `GfxBase` is invalid and unsupported.
2. Although there is still the notion of PAL-machine vs. NTSC-machine, the new display modes support PAL and NTSC screens on any machine. Also, do not assume that if the PAL bit is clear, the NTSC bit must be set. Both bits clear is reserved for future use. If you find both bits clear, the recommended processing is to exit gracefully.
3. Text rendering now handles `ColorFonts` so there is no need to install the `ColorFont` wedge under V2.0.
4. Text rendering is now much faster so there is no need to install `FastFonts` under V2.0.

V. Changes to intuition.library

1. Intuition has a new state machine. This should fix all of the problems with gadget activation, window events, and user mouse clicks. For example, closing a window while a string gadget is active in it, will no longer leave the system in a state of confusion. Operations will now correctly block while the objects for that operation are in use. Some of the really unusual work-arounds may no longer be needed and could, in strange cases, modify the behavior of your software.

2. One of the more complex aspects of the new state machine is that of MENUVERIFY deadlock avoidance. (Note that in all of this, only messages you have IDCMP flags set for will be delivered.) MENUVERIFY will be aborted in if any one of the following happen:
 - a. Preferences Verify Timeout elapses, that is, you have not responded to the MENUVERIFY event within that time period.
 - b. The user releases the menu button before you respond to the MENUVERIFY event
 - c. You call ModifyIDCMP() before replying the MENUVERIFY event. This usually means that you have not yet seen the event.
 - d. Intuition can not send the MENUVERIFY event.
3. When a MENUVERIFY is aborted since a message can not be removed from your message port, the MENUVERIFY message will still be on your port. A MENU PICK event will be sent that has MENUNULL. If it was due to the user letting go of the menu button, a MENUUP event will also be sent.
4. Messages you will get (compared with V1.3) after you respond to the MENUVERIFY message:
 - a. If you MENUCANCEL after Intuition had timed out, you will get an unexpected MENU PICK/MENUNULL before you get the expected MENUUP.
 - b. If the user lets go of the menu button before Intuition hears back from you or it times out, you will receive a MENUUP followed by a MENU PICK/MENUNULL whether you had responded MENUCANCEL or MENUHOT to the MENUVERIFY message.

5. Software needs to be able to handle the improved flexibility that 2.0 has given the average user. Software must be able to deal with various system fonts. Font changes are more global and selection of default fonts is dangerous if you did not plan for it. Using NULL for TextAttrs in IntuiText structures could lead to surprises. Likewise, not checking the font that was set in the RastPort could lead to surprises.
6. Do not assume that the window or screen title bars are of a preset size. The size is determined based on the font that would be used in the title area.
7. Software must be able to deal with various Workbench screen resolutions and depths. The Workbench screen now can be set to more than just 640x200 or 640x400. Resolutions may be anything from 640x200 on up. Display depth can now range from 1 bitplane to the maximum available in the display mode.
8. The parts of a window that are in the border are now fully claimed by Intuition. This includes the thin, color-0 line which is inside the color-1 (or BlockPen) line. It also includes the area in which there are any border gadgets such as the area above the sizing gadget (when the sizing gadget is in the right border).
9. The border sizes for the window have always been in the Window structure. These are the pixel sizes of the border of the window. They are:
 - a. Window->BorderLeft
 - b. Window->BorderTop
 - c. Window->BorderRight
 - d. Window->BorderBottom
10. Gadgets that were in the window border area should have had the LEFT/RIGHT/TOP/BOTTOM xxxxBORDER flag set. Due to changes in window border rendering by the system, any software that renders into the window borders directly may be in for a surprise. The best solution is not to render within the border areas of a window.

11. Detail and block pens are no longer used for the window rendering. They are still used for the menus. Custom screens that do not use the new OpenScreenTags function of 2.0 will get a "monochrome" mode for the windows on that screen. Windows on the Workbench screen (or other screens opened in the new 2.0 manner) will use the DrawInfo from the screen to determine the rendering method.

VI. Changes to diskfont.library

1. It will now try harder to match the style flags you ask for while at the same time trying to reduce disk access.
2. The .font file format has changed. Although the change is minor, anyone who is messing with the .font files directly may need to be fixed. Since the format is different, the new .font files have a different FCH_ID, too. The format change is at the end of each font entry. Some room at the end of the font name character array is now used for some new data. The FCH_ID on "compatible" font files will change only in the last nibble (lowest 4 bits) of the ID.
3. It is very important that you only set the flags you really want when opening a font. If you used -1 as the flags (all bits set) or had trash values in the flag fields, you might get surprises. Bitmap font scaling is one of these new flags. Thus, if you ask for a size that does not exist and you have the scaling flag set, it will scale.
4. If you ask for a font, be prepared to handle it. Don't ask for an arbitrarily large font in an attempt to get the largest. If the font doesn't exist in the size that you specify, the system will create it.

VII. Changes to dos.library

1. The dos.library is now in C and uses normal stacks. The "strange" BCPL actions are gone.
2. Under V1.3, dos.library return values were passed in d1 as well as d0 due to a quirk of BCPL. If your code relied on the return in d1, it will break. For example, under 1.3 a call to Lock() would return the result in d0 (and d1). UnLock() needs its argument in d1 - if you forget to move it from d0 into d1 after the Lock() call your code will work under 1.3 and fail under 2.0. Due to the number of programs that run into this problem, there may be a short-term patch for this.

3. The dos.library calls should not be made from a task. Some of the calls that seemed to work before are now unsafe.

VIII. Changes to ramlib (that which does OpenLibrary/Device magic)

1. OpenLibrary() checks LIBS: and the :LIBS/ directory on all devices. This should make using a program with custom libraries much more of a "Just put the disk into the drive and double click"
2. OpenDevice() checks DEVS: and the :DEVS/ directory on all devices. This should make using a program with custom devices much more of a "Just put the disk into the drive and double click"
3. Expunge is a bit smarter in that it will make another pass over the library list to find any library that was not expungeable before but is now. This happens when a library or device that is expunged closes another library that could now be expunged.

IX. Changes to timer.device

1. It now correctly allocates all of CIA-A. This means that any program that tries to allocate CIA-A will fail.
2. The time values in the timerequest structure now come back as garbage. It was always documented that these values could be invalid after the timerequest comes back - now they are. This did not happen, in most cases, under V1.3. Software which does not reset the fields between timerequests will break.
3. Timer events do not have to come back in the same order as sent. In 1.3, events that were to come back at the same time (or at almost the same time), were returned in the same order in which they were sent. This was never guaranteed, and in 2.0 it may not happen.

Most of the details on the new Zorro III expansion bus was talked about by Dave Haynie in his talk on the Zorro III Bus Specification. It has a section on compatibility with the Zorro II bus. I will touch on a few of the points.

X. Changes to the hardware in the A3000 and Zorro III bus

1. The Amiga 3000 bus controller emulates, for the Zorro II bus, the signals of the published Motorola 68000 specifications. Anything that is designed based on these specifications should work. Things that are designed based on observed 68000 behavior rather than the documented 68000 operation is at serious risk.
2. Some of the 68000 emulation control lines are gone. New signals are now on the VPA and VMA lines.
3. The bus is better terminated. This should mean much less noise on the bus. If your card's line drivers are too "light" they may not be able to drive the A3000 bus.
4. The A3000 Buster does fair bus arbitration, which means that if you request the bus, you will get it at some point. Starvation is now impossible. Also, slot position no longer effects the speed at which bus grants are given.
5. Cycles that are not in the Zorro II address space do not generate a Zorro II cycle. This lets the system run faster than the limits of Zorro II when accessing address ranges that are in the 68000 space but not Zorro II space. For example, access to CHIP memory will not be visible on the Zorro II bus. Since there was no legal way to use any of this on the bus, this should not cause any problems.
6. A new 1.5 megabyte area for I/O devices that is mapped to \$00A0 0000 to \$00B7 FFFF.



—

—

—



Standards

By
Paul Higginbottom
Manager, Developer Support

Purpose

The purpose of this document is to describe the current state of application standards for the Amiga and to outline some of Commodore's exciting plans in this area, and the cooperation we need from you, our valued developers.

What are standards?

Standards are guidelines for the actions of members of a group. However, the benefits of complying with standards must be explained in order for them to want to do so.

While the computer industry has many different standards, the idea of standards is not new, and applies to all systems, including societies—e.g., moral standards.

Now that we know what standards are, what are the benefits of conforming to them?

Standards promote consistency

Without standards there would be chaos or anarchy. Imagine life where every member of a community does their own thing, without regard to the impact on the community as a whole.

Standards create an environment with fewer intangibles and greater predictability. This makes any system easier to understand and live within.

A concrete example of why standards are important would be traffic lights. Everyone knows that red means stop and green means go. Imagine the outcome if those colors weren't standard. Imagine if in one place they decided that red means go and green means stop!

Standards promote faster progress

If you adhere to a standard, then enhancing the standard as well as conforming to new standards is much easier because only one set of guidelines is required for upgrading to that new standard.

In the absence of any standards, progress is very difficult because each group member requires an individual plan.

Standards and you

You may be asking: 'Why should I adopt standards?,' or more specifically: 'What are the benefits?' Adopting standards will result in:

1. Giving your product a wider appeal—more sales—because of its combination of familiar and unique features, its standard user interface, and its degree of compatibility with the system and other products.
2. Reducing your support costs—more profits—because users will not be confused by a non-conforming user interface or features, or the product's incompatibility with other ones. Remember, no software is an island!
3. Increased credibility for the Amiga as a whole, and therefore greater appeal and a larger installed base to sell to.

Background

The quantity and quality of available Amiga software and hardware products has increased tremendously since the introduction of our first Amiga model—the A1000—in 1985.

Unfortunately, there is a great deal of inconsistency in the features and operation of these products which often results in confusion for the end-user. In addition, the lack of standards has meant that many products do not work well together.

User interfaces vary widely, ranging from primitive and inflexible; creative but unusable; incredibly flexible but confusing; to professional and useful. A standard way for users to operate programs and hardware is essential if the Amiga is to become successful as a mainstream player. Computer purchasers today—businesses in particular—buy computers to make people more productive, and they will often buy the system based upon the fact that it requires the least amount of training and expertise to operate.

Amiga vs the competition

The two major competitive platforms for the Amiga in mainstream computing applications are the Apple Macintosh and all of the various PC compatibles. Consider the state of standards on those platforms:

- *Macintosh* - The Macintosh has become successful in large part because of its rigid and standardized operating environment and application software that is very consistent.
- *PCs* - There are a vast range of products available for PCs, and the emerging user interface standards are Windows under MS-DOS and Presentation Manager under OS/2. However, the majority of PC users have not switched to either environment yet.

Amiga standards and tools

It should be clear by now that standards and compliance with them have a very large influence on the success of any system. What follows is an overview of the current Amiga standards and examples of their use.

IFF

Proably the most well defined standard on the Amiga, the interchange file format (IFF) allows users to create files in one software package and then use those files in another. The most widespread use is with bitmapped graphics files where a user might digitize a picture into an IFF file with one package and then place the digitized picture in a document with a desktop publishing program.

There are other types of IFF files, including formats for audio sounds, music, animation, and text. IFF is the multimedia file format for the Amiga.

Clipboard

The clipboard device is—unfortunately—a very under-utilized feature of the Amiga operating system. We have published articles on how to incorporate it into your programs. This feature allows the passive transfer of data by the user between programs that have no knowledge of one another. For example, a user could copy some data in a spreadsheet to the clipboard, and paste it into their word processing program. This is not the same as active transferring of data—often referred to as ‘hot-links.’ The latter can be implemented through the new file notification features of 2.0.

AmigaDOS 2.0

2.0 is mentioned here because many of the utilities, tools, and Workbench are excellent examples of the standard look and feel.

At the system level, the following tools are available to allow you to quickly and more easily implement a standard look and feel:

1. GadTools—the gadget and menu layout toolkit.
2. IFFParse.library—toolkit to parse IFF files.
3. ARexx—macro language provides inter-process communication.
4. ASL.library—standard file, and font, and ‘list’ requesters.

User interface style

In the *ROM Kernel Libraries and Devices Reference* manual are style guidelines (p211–220) to assist you in creating applications with a standard look and feel.

AmigaVision

Included with most Amiga models now is AmigaVision, our icon-based authoring language. AmigaVision allows users and developers to easily create multimedia presentations and computer-based training courses.

AmigaVision is mentioned here because it represents a good example of look and feel, and also runs under 1.3 and 2.0, taking advantage of 2.0 if it is available. AmigaVision is not designed exactly according to the 2.0 look and feel, but should stimulate some ideas.

What's next?

At Commodore, we realize that it is our job to coordinate the development of standards and to encourage you to adopt them.

In addition to the current standards outlined above, we are working on the development of additional standards and tools to make it easier for you to create software, and to increase the appeal of your products.

The two subsections on AppShell and AppBuilder deserve special mention. They are tools we are working on that will make it very easy for you to build the standard 'look and feel' into your products. Read these sections and other related sections in the 1990 DevCon Notes binder. An alpha version of AppShell is included on the DevCon disks.

Look and feel guide

A complete style guide is under development that will define in detail a standard look and feel for applications.

Application user interface

Here is a list of standards to consider when designing an application user interface:

1. An application's user interface should consist of more than menus and gadgets.

For example, it is important that a full ARexx interface be available so that applications can be driven from scripts and from a command prompt either in the shell or one built into the program. It is also important that applications support the new Workbench object-oriented functions (AppIcon and AppWindow) to allow more intuitive manipulation of projects and actions by the user.

2. Separate application functions and their means of invocation.

Each type of user interface—keystrokes, menus, gadgets, ARexx—is another way to invoke the features and functions in your applications.

The most important design goal you should adopt is the separation of application functions and the means of invoking them. Too often a function is designed to fit the user interface and not the other way around. For example, saving a file should not be thought of in terms of the 'Save' and 'Save As...' commands in a 'Project' menu.

3. Applications should adhere to system preferences by default.

Some of us may like certain colors, fonts, screen resolutions, and other details, but please do not force them on the user. Use system preferences by default, and also allow the user to customize and save the application environment.

4. Applications should support both keyboard and mouse interfaces.

No matter what preference you might have as to whether applications should be mouse or keyboard driven, both should be implemented. Mouse operation tends to help and attract new users immensely. Power users on the other hand, often prefer keyboard operation.

5. A final note about application user interfaces...

Remember that a user is trying to get work done with your products.

The user interface of a product should make getting from point A to point B as short a journey as possible.

AppShell

The AppShell is a skeletal application we have developed that implements handlers for all of the types of user interface (as outlined above) including:

1. *Workbench* - Supports the new AppIcon and AppWindow features.
2. *Preferences* - Adapts to system preferences unless application-specific preferences are present.
3. *Keyboard input* - Processes any and all keyboard input.
4. *Intuition input* - Processes all Intuition input.
5. *ARexx* - Processes all ARexx interaction.
6. *SIPC* - Simple Inter-Process Communication. This is AppShell specific, and is provided for fast communication between processes.
7. *Asynchronous processes* - Allows an application to easily manage multiple projects, tools, and other processes such as printing.

Implemented as a link library and run-time library, AppShell will allow memory requirements for multiple applications to become lower through common code. In addition, if you create your application using AppShell, you will get all of the above features basically for 'free'.

This tool will be extremely valuable, particularly in the development of new applications you may be planning. While you may not wish to abandon a code base for an existing product, you may wish to use AppShell to create an upgrade of an existing product as it will be the fastest way to implement all of the features mentioned above.

This project has been coordinated by David Junod, and his notes on this valuable tool are included in the 1990 DevCon Notes binder.

AppBuilder

The AppBuilder is a tool that is designed to make it quick and easy for you to design an application user interface according to the guidelines outlined above.

More than just designing menus and requesters though, AppBuilder allows the correlation of application functions with their multiple means of invocation through menus, buttons, keystroke combinations, ARexx commands, etc.

AppBuilder will be able to output C or assembler source code, or BLINK compatible object code.

AppBuilder will be able to generate data specifically designed to work with the AppShell if you so choose. With this option enabled, application development can become as simple as:

1. Specify the functions within an application and their names using a word processor or outline processor.
2. Specify the means of invocation for the application functions and other user interface details with AppBuilder. Outputs object code, for example.
3. Write and compile your application functions, which generates object code.
4. Link AppBuilder output, application function object code, and AppShell link library to create finished application.
5. Test, and loop to any of the previous steps.

Multimedia

Some time ago Commodore began looking at developing a MIDI standard for the Amiga. Such a standard would consist of a software driver that multiple MIDI applications could utilize to share common resources such as the serial port.

We have realized that the MIDI issue is actually part of a much larger issue, namely the synchronization and allocation of resources for programs using a variety of media, including—but not limited to—Amiga graphics, Amiga audio, MIDI, and SMPTE control of audio and video.

We are investigating the possibility of developing a global synchronization scheme for multimedia. This would allow different applications to synchronize together for presentation, training, video production and audio production uses. Such a standard would truly increase the versatility of the Amiga.

A global synchronizer would provide precise, high resolution timing to clients, as well as high speed I/O for data processing. For example, MIDI events would be initially processed by a handler and then made available to a chain of servers much in the same way interrupts are handled under Exec.

In addition to global synchronization, there is the problem of resource allocation. For example, a laser disc player needs a serial port. So does a touch screen. MIDI control requires a serial port, but not just any one, it has to be one that can support 31250 baud. SMPTE may be hooked up to a serial port. A sound digitizer may be connected to a parallel port. There might be other controllers or controlled devices being accessed through game ports, yet more serial ports, or even disk drive ports or expansion slots.

Constraining the discussion to ports for now, we are also investigating the idea of some kind of I/O port management scheme, whereby ports could be allocated to specific needs and could be permanently configured as such via a Preferences type of scheme. For example, a person with a full blown multimedia set-up probably is unlikely to change the port uses once they have them configured. However, a system with multiple serial ports for example, cannot have system software which allocates ports on a first come, first serve basis à la the audio device. We need I/O port management and are investigating it.

Hard disk installation

One area that desperately needs a standard is the way in which applications can be installed onto hard drives from floppy disks.

We are working on an install utility that will be available to you to customize (while retaining a standard look and feel) and include with your own products.

In addition to just moving files from distribution floppy disks to a hard disk, an install utility must be capable of reading and modifying disk icons, scripts, and configuration files.

Networking

As you probably know, Dale Luck has been coordinating a standard networking software layer for the Amiga. This will allow the seamless integration of multiple networking protocols, even using the same Ethernet network adaptor, for example.

This is a key step toward the full potential of the Amiga being realized.

Where do you begin?

It is our sincere hope that you will work with discipline and speed to make your products conform to current and proposed standards. We are working hard to make sure that you have examples, tools, documentation to make the journey easier.

So as the title says, where do you begin? You might begin by making sure the Intuition portion of your programs is standard. Take a look at the Preferences editors under 2.0. Take a look at AmigaVision. These are examples to model your products on. By using the tools available under 2.0 such as GadTools and ASL.library, you can be assured of a standard look and feel for many parts of your program.

Once a standard Intuition look and feel has been accomplished, please add ARexx to your programs so that they may work more easily with other applications. Support the clipboard. Support Workbench. The list goes on, but the benefits are enormous.

Summary

As was mentioned at the beginning, the purpose of this document was to describe the current state of application standards for the Amiga and to outline some of Commodore's exciting plans in this area which can definitely increase your company's bottom line.

I hope you are as excited as I am, and agree that we are doing the right things to help the Amiga mature into a powerful AND easy to use platform. We are far from complacent about our mission though, and we want feedback from you—positive and negative—about our strategy and tactics.

One last note which is worth repeating: A standard way for users to operate programs and hardware is essential if the Amiga is to become successful as a mainstream player.

—

—

—



CTracTM Emulation System for CDTV

by ICOM Simulations, Inc.

The CTrac Emulation System for the Commodore CDTV is a product that emulates the CD drive of CDTV for the purpose of providing a developer environment capable of making rapid changes to a CD-based product. Because of the inherent read-only nature of CD, and the specific access times and transfer rates of CD data, a high quality emulation system is necessary to allow rapid and accurate development of CD titles. CTrac is designed to accomplish that task.

The CTrac Emulation System consists of two parts. The first part is a printed circuit card that plugs into an Amiga 2500/30 or 3000 and its associated driver software, which is known as CTrac Emulator. The second part is the software to create CD-ROM disc images, which is known as CTrac Builder.

The CTrac Emulator, along with its driver software running on the above mentioned Amiga 2500/30 emulates the CD-ROM drive of the CDTV so that a program sees no difference between CDTV with an emulation board and a normal CDTV. It does this by completely replicating all aspects of the CD subsystem, including recognizing and handling all commands to the CD, reading data and the proper transfer rates, emulating CD seek time and rotational latency and all other timing characteristics of the CD. It is this level of transparency that makes true emulation, rather than approximate simulation, possible. CTrac is capable of handling CD-ROM data, CD-DA (digital audio) data, CD+G data, and subcode channel data, all coming from a properly built disk image residing on the Amiga 2500/30 or 3000.

The CTrac Emulator driver software is responsible for interpreting the commands sent to the CD-ROM and converting them to disk commands to fetch the required data from the disk image residing on the development system's hard disk. It will then transfer the data to the CTrac Emulator so it could be fed at the proper data rate to the CDTV. It would also be responsible for emulating the seek time that the CD-ROM requires so as to make the total emulation as accurate as possible.

The second part of the CTrac Emulation System is CTrac Builder, which allows a developer to take all of the portions of his product, including program files, data files, and CD-DA data (if any) and combine them into a single image of a CD-ROM disk. This is exactly equivalent to creating a master image from which an actual CD-ROM disk could be manufactured. The difference is that the image is created such a manner that the data is ready to be fed to the CTrac Emulator so it can be sent to the CDTV as if it came from a CD-ROM disk. CTrac Builder allows a developer to group his files in any manner necessary, and is capable of creating multi-track disk images. CTrac Builder does all the calculations need to provide a complete disk image ready for testing, including the ISO-9660 disk format, as well as ECC data, data scramble, subcode date, and CD+G data.

The CTrac Builder is actually multiple programs. The last program is the image builder, which takes in various types of data tracks and outputs a disc image. It is controlled by an Image Description file, which will contain a series of commands describing which data files will be used as source material for the tracks of the image, as well as in what order they need to be included and what, if any, special processing will be needed by a particular track. This Image Description file will be a simple text file, which will be generated by the track builders. The track builders make up the rest of CTrac Builder. They are responsible for creating the individual tracks needed by the disk, and which are used as input for the image builder. The track builders will be controlled by a Disk Description, which provides information about the disc as a whole, as well as the individual tracks that are part of the disc.

Commonly asked questions about the CTrac Emulation System:

Q: Will the Ctrac system allow me a way to transfer my disc image to a CD manufacturer to have test discs and final discs made?

A: Yes. It is intended that CTrac will support methods of data input to CD manufacturers. The first will probably be the Exobyte 8 Track tape recorder, and the second will be a Write-Once CD system. Further details will be announced in the near future.

Q: Do I have to take my data to a mastering facility so that I can format it properly as an ISO 9660 CD?

A: No CTrac Builder functions as a complete mastering facility all by itself. No additional steps are needed.

Q: Now that I can test my CD program with CTrac Emulator, how will I be able to create and test CD Audio (CD-DA) tracks?

A: The CTrac System is fully capable preparing and emulating CD Audio. (CD Audio is also known as CD-DA or Red Book) in fact, you could use CTrac to make Red Book compatible audio discs as well as CD-ROMS.

Q: What size hard disk will I have to have to use the CTrac Emulation System?

A: CTrac does not require specific hardware configurations. The size of the emulation disk, therefore, depends on the size of your project. We usually recommend at least twice as much room as will be filled on the final disc. For example, if you are creating a product that will occupy 280 MBytes of data on the CD, a 650 MByte disk should be sufficient.

Q: How will I tell the CDTV that I am using an emulator. Will I have to run a program on the CDTV to get the emulation started?

A: No, the emulation is completely transparent to the CDTV. It cannot tell that an emulator has been substituted for its CD-ROM drive. Just start the emulation driver on the Amiga 2500/30, insert the disk to be emulating by opening the disk image file in the driver, and reboot the CDTV. It will start up just as if you had inserted a real CD in its CD-ROM drive.

For more information, contact Mike Kawahara or Rick Unland at Commodore Business Machines, Inc., or call ICOM Simulations, Inc. at (708) 520-4440.

Getting The Best Image For Your CDTV Application

by Perry Kivolowitz, ASDG, Inc.

Copyright 1990 By ASDG Incorporated - All Rights Reserved

1 Overview

In this article, I will present issues which developers should be aware of when generating images for use with the CDTV system. I will also present several techniques for maximizing image quality making use of ASDG's The Art Department.

2 General Issues

Following is a discussion of issues which relate to displaying imagery on any Amiga based system.

2.1 Choice Of Video Mode

The choice of video mode is one of the most basic decisions you will have to make when preparing an Amiga based application. The choices which you must make relate to:

Resolution

The Amiga offers high and low resolution modes. Each offers some advantages but at some cost.

High resolution offers a sharper image and permits more information to be displayed at one time. However, it limits the number of colors displayable at one time and can degrade system performance.

Low resolution offers a richer color capability, can use less memory and does not impact system performance as much as high resolution. However, low resolution screens cannot display as much information and can appear "chunky."

Palette Depth

The number of colors which will be used during the display of images must also be considered. A richer set of colors nearly always produces more pleasing results. However, deeper palettes come at the expense of resolution (you must forego high resolution if you want more than 16 colors), memory, and animation speed.

Interlace

The vertical resolution of an image can be doubled by placing the Amiga into an interlaced video mode. Interlaced video does not increase the loading caused by the display upon the processor but it does double the memory requirements of any given display.

The principal liability of interlaced video is the visible flickering that it can cause. Horizontal lines, especially when thin and highly contrasting to surrounding pixels, can produce an extremely noticable flicker which can become annoying.

Overscan

By using overscan, the visible border around an Amiga screen becomes usable display area. Overscan (which can be along the vertical and horizontal dimensions independently) adds to the TV like appearance of an Amiga display. An overscanned screen consumes more memory and can drain a significant amount of processor speed especially in some high resolution screen modes.

The degree of overscan which is necessary or acceptable varies depending upon whom you ask. Many people specify a maximum overscan screen as 768 pixels wide (in high resolution) by 484 high pixels (in interlace) in NTSC or 592 pixels high in PAL. However, we recommend a limit of 736 pixels wide in high resolution.

2.2 Mixing Video Modes

On the Amiga, it is possible to mix different video modes on-screen simultaneously with the following restrictions:

- Different video modes may be horizontally stacked only. Two video modes cannot share the same scanline.
- A small number of lines (approximatley 3 non-interlaced lines) become unusable when transitioning from one video mode to another. Worse still, the mouse cursor will become obscured while passing through these lines.
- Transitioning from one video mode to another consumes a small amount of processor bandwidth.

- If any of the visible screen modes are interlaced, the entire screen will be displayed in interlace.

Within these restrictions, you can see that it is possible to create a display with (for example) text displayed in high resolution and four colors and an image displayed in low resolution with 64 colors.

2.3 CHIP Ram Contention

Some Amiga display modes place a greater load on machine resources than others. The following tables indicate a relative amount that a given display mode (without overscan) will impact the CPU when it attempts to gain access to a CHIP memory location:

Horizontal Resolution	Number Of Bit-Planes					
	1	2	3	4	5	6
Low	None	None	None	None	Some	Moderate
High	None	None	Moderate	Considerable	-	-

Overscan can considerably decrease available processor time. When displaying a high resolution four bit-plane screen, for example, the processor is locked out from accessing CHIP ram during the display of an entire scanline. It can only access CHIP ram during the horizontal and vertical retrace times. Horizontal overscan shortens the available horizontal retrace time and vertical overscan further diminishes the length of available vertical retrace time.

Note that the processor is not impeded from accessing memory located on the FAST memory bus.

2.4 Memory Requirements

Each screen mode will consume a different amount of CHIP memory. The following tables indicate how much memory is used (in bytes) for each of the given common screen formats:

Screen Size	Number Of Bit-Planes					
	1	2	3	4	5	6
320 by 200	8000	16000	24000	32000	40000	48000
320 by 400	16000	32000	48000	64000	80000	96000
640 by 200	16000	32000	48000	64000	-	-
640 by 400	32000	64000	96000	128000	-	-
368 by 240	11040	22080	33120	44160	55200	66240
736 by 480	44160	88320	132480	176640	-	-

2.5 Aspect Ratio

Each dot displayed on an Amiga screen has a specific shape. Unfortunately, this shape is not square. As a consequence, pixel aspect must be considered when displaying images from a wide variety of sources.

Many factors affect the aspect of on-screen pixels. These include:

Screen Format

Switching between high and low resolution doubles or halves the number of pixels displayed across the screen. Clearly, this affects the width of each pixel with high resolution pixels being half as wide as their low resolution counterparts. Similarly, switching between interlaced and non-interlaced video halves or doubles the number pixels shown vertically.

Dots on the Amiga screen most closely approach square when in:

- Low Resolution, Non-Interlaced
- High Resolution, Interlaced

In these modes, the ratio of a pixel's width to its height is approximately 10 to 11 in NTSC. Low resolution pixels shown in interlaced video are approximately 5 to 11 (width to height).

NTSC Or PAL

PAL video fits more horizontal lines onto the same sized display area. Therefore, PAL systems can come closer to square pixels than NTSC systems.

Monitor Settings

Every monitor has controls (often accessible by the user) which affect the width and height of the displayed image. Clearly, even the most careful planning and compensation for pixel aspect can be undone by the user.

Without considerations for pixel aspect, images scanned with most optical scanners (which produce square pixels) will appear stretched or distorted even when shown in low-res/non-interlaced or hi-res/interlaced modes.

Other sources of square pixels include 3D modeling programs and images created on non-Amiga computer systems.

2.6 International Video Formats

As indicated in the previous section, the differences between NTSC and PAL will affect your product development by affecting the aspect of displayed pixels. NTSC and PAL

also differ in how much displayable area is available on-screen. The standard height of a non-interlaced, non-overscanned NTSC screen is 200 pixels. The same screen on a PAL system will be 256 pixels high.

The problem that different imaging areas present is particularly nasty if you must have only one set of images for usage on both NTSC and PAL systems. For example, if an image is created for proper viewing on an NTSC system, it will not fill as much of the screen and will appear distorted on a PAL machine. If an image is intended for proper viewing on an PAL system, then part of the image will be obscured on an NTSC system.

Therefore we suggest that requiring only one set of imagery for use on both PAL and NTSC systems may be an unrealistic goal. We recommend the creation of a separate set of images for use on PAL and NTSC systems. Since CD-ROMs are quite large, and other internationalization factors will come into play in your software anyway, this may not be too unpleasant.

3 CDTV Specific Issues

This section contains a discussion of issues which apply specifically to images for use on the CDTV system.

3.1 TV's Not Monitors

As a developer, you can expect the Amiga owner to have a high resolution computer monitor for use on their system. This is not the case with CDTV. The expected display device is an ordinary television. This impacts you in two ways:

1. First, televisions vary incredibly in quality and sharpness. High resolution text (for example) which is perfectly readable on your development system may not be readable at all on a television.
2. Second, the interface to television, composit RGB, is not as precise as the analog RGB normally used in Amiga displays. High contrast transitions which appear perfectly on your development system may become ugly masses of bleeding color on a television.

Our recommendations include:

- No CDTV development environment is complete without a *cheap* color TV running in parallel with your high resolution computer monitor. The best way to anticipate how your product will look in the consumer's home is to view your product the same way the consumer will.

- Avoid high contrast transitions especially where text is concerned. Especially avoid saturated reds.
- Avoid images which are too bright. Specifically, try to keep your brightest colors at an intensity of less than 13 (using the Amiga standard scale of 0 to 15).

3.2 Distance

The typical computer user views his high resolution computer monitor from no more than four feet. The typical television viewer is generally 6 to 10 feet from the set. This means that visual detail may be lost simply because the user is further away from the display device than you had anticipated.

This can be a significant advantage:

- The advantages of dithering become even more pronounced as the viewing distance makes it unlikely that the user will be able to discern the individual dots.
- Low resolution displays (with their richer color palettes) can be much more effective than limited palette high resolution displays. The larger viewing distance means that richness of color will be much more important than sharpness of dots.

3.3 Phosphor Burnout

It is very likely that a CDTV user will leave the device on for long periods of time without actually being present to cause the screen display to change. If your application does not include a self contained "screen blanker" it is likely that your application will burn a hole in the user's television set, not something which is likely to please your customer.

4 Getting The Best Image

This section describes various techniques which are helpful in getting the best image quality possible for your CDTV images. These techniques assume the use of ASDG's The Art Department (TAD) for image development.

4.1 Image Sources

It is possible that no personal computer has ever offered more alternatives for how to capture or generate images as the Amiga. How you generate or capture an image can significantly affect the quality of your end product. Following is a summary of imagery sources and where we recommend their use:

Color Scanners

Color scanners should be used whenever you need to capture flat art. Video digitizers simply do not have the resolution to provide high quality imagery for a broad range of applications. We do not recommend purely gray scale scanners for CDTV applications since CDTV is primarily a color device. Additionally, color scanners can also scan in gray scale and cost only marginally more.

Color scanners also offer advantages in speed and ease of use compared to other color image capture systems.

Video Digitizers

We recommend video digitizers for capturing non-flat art or for capturing images from a live or video source. The quality of the images you can capture with a video digitizer is strongly affected by the quality of your video source. Specifically, where a video camera is being used, the quality of the camera can make or break the image.

Paint Boxes or Paint Programs

There are a number of high-end paint boxes which can create enormously detailed imagery such as the Quantel and Wavefront systems. Also, images created with any personal computer based paint programs such as Deluxe Paint, ColorRIX, TIPS, or RIO can be employed.

3D Modeling Systems

Three-D modeling systems can play an essential role in image development. This is especially true for the creation of complex scenes which cannot be scanned or digitized since they do not actually exist in the real world.

4.2 Dithering

Dithering is one of the most important techniques for increasing visual realism in your CDTV images. Dithering sacrifices some of the spatial sharpness of the image to dramatically increase the color fidelity of the image.

Carried to an extreme, dithering can produce the appearance of true gray scales on a single bit-plane monochrome screen. More typically, the dithering techniques found in TAD can produce the impression of hundreds of colors on a 16 color screen, or the impression of many thousands of colors on an Extra-Half-Bright or Hold-And-Modify screen.

TAD offers 7 dithering methods (including the choice of no dithering). We have found our Floyd-Steinberg implementation to be suitable in most instances, especially when creating images for display on high resolution screens.

Dithering cannot increase color fidelity where no increase is possible. For instance, there is no point in dithering a 32 color picture, if the original image data contained only 32

colors. In other words, for dithering to have an effect, there must be more colors in the original data than there will be in the rendered image.

TAD can synthesize new colors, however, when its scaling function is used. For example, if a 640 by 400 32 color image is scaled down to 320 by 200, TAD can synthesize as many as thousands of new colors as it performs the reduction. This is accomplished by pixel averaging all the different combinations of the original colors. In this way, the spatial resolution lost in reducing the size of an image can often be compensated for with increased color range.

Dithering can sometimes produce unwanted seemingly stray dots. These can be eliminated within TAD by slightly increasing the contrast and re-rendering or by invoking the RIP (Remove Isolated Pixels) function.

4.3 Aspect Correction

TAD offers highly precise scaling both upwards and downwards in size. When correcting for aspect, don't forget that the width can be enlarged rather than always shrinking the height.

An easy way to determine just how far off-square your Amiga/Monitor combination is, is to draw a 1 inch square on paper and then scan it in. Using TAD, display the alleged square in many diverse screen formats and experiment with the scaling functions to gain experience on making a square, square.

4.4 Interlace

4.4.1 Making An Interlaced Image

Interlaced low-resolution images, especially in HAM, can appear exceedingly crisp on a CDTV display. Interlaced low-resolution is, however, one of those resolution combinations which are nowhere near square in aspect.

To produce an interlaced low-resolution image, simply take your square or near-square aspect image and scale down the width by approximately 50 percent. Alternatively, you could scale up the height by 100 percent.

4.4.2 When To Interlace

Natural images (people, places, etc.) can generally be displayed in interlace without significant flicker problems. Images which have a lot of horizontal lines or have very stark transitions from color to color (such as some images created with 3D graphics programs) will fare very poorly when displayed in interlace.

4.4.3 Overcoming Interlace Flicker

If you have a problem image which you need to display in interlace, try the following:

- Try reducing the contrast of the image slightly. This may cause any flickering scanlines to become more subdued and therefore less noticeable.
- Try scaling the image upwards or downwards slightly. This may cause flickering scanlines to be spread over more or fewer displayed scanlines and therefore be less noticeable.

4.5 Increasing Visual Punch

4.5.1 Contrast

As indicated in the preceding section, a slight increase in contrast can eliminate seeming stray dots when rendering a dithered image. A slight increase in contrast can also add a considerable amount of visual punch to an image.

4.5.2 Gamma Correction

TAD offers variable Gamma Correction (non-linear color correction) which allows you to brighten an image without loss of detail which the standard brightness control would cause. Increasing the Gamma value of an image can dramatically increase the visual punch of an image, can be used as a special effect, or can bring out detail in a dark image.

4.6 Who's Afraid Of Gray Scale

Sixteen shades of gray (especially when dithered by TAD) covers the spectrum of grays far better than even 64 or 4096 colors covers the spectrum of color. Don't be afraid of using gray scale images in your application.

TAD offers a color to gray scale conversion function which takes into account the relative frequency response of the human eye. The color to gray conversion function can produce some exceedingly realistic images.

4.7 Flips And Mirrors

If your subject matter allows for flips and mirror images, you can increase your composition flexibility by using a flip or mirror of an image rather than the original image. For example, if a person looking off to the left simply looks better than a person looking off to the right, flip him.

Avoid flips or mirrors when text is visible in the image or when displaying technical data or drawings when flipping would either be noticeable or change the meaning of the image.

4.8 Genlock Considerations

It is possible that your application may be used on a CDTV on which there is a Genlock device. In this case, Genlocked video will appear through any areas in your imagery which are drawn in pen (or color register) 0.

You can easily make images Genlock opaque in TAD by instructing TAD not to use color register 0 during its rendering. This is accomplished in the Palette control panel by selecting a non-zero color offset and requesting a CUST (or custom) rendering.

4.9 Mixing Computer Chosen And Manually Chosen Colors

If you wish to render text directly over an image you may wish to take the text colors into consideration when producing the palette for the image. If the text colors are not taken into account, then you will be forced to use one of the colors appearing in the image. This often produces unacceptable results.

Using TAD's palette controls, you can set aside color registers to be used for titling in two ways.

The first method produces images which do not have the reserved color registers appearing anywhere in them. This can be done using the same technique as reserving color register zero described in the previous section. Simply decrease the number of colors to be used and set the offset of color zero to the desired value.

For example, to reserve four colors for titling within a 32 color image:

1. Set the total number of colors to 32. This defines the depth of the resulting image.
2. Set the number of colors to be used to be 28.
3. Set the offset of color zero to either 0 or 4. Setting this value to zero reserves a block of 4 colors after the 28 colors used by the image. Setting this value to 4 reserves 4 colors prior to the 28 used by the image.
4. Render the image using the CUST setting.
5. Set the 4 reserved colors to their desired values.
6. Save the image to disk.

This will produce an image which will not contain any reference to the unused block of registers. You can modify the unused color registers to any value and not affect the look of the image.

The second method allows you to set aside a given number of colors as before, but then allows you to merge these reserved colors into the overall image. This allows you to specify specific colors which *must* be present in the image, and then lets the Art Department do the rest.

Jumping into the above sequence of steps at step 6:

6. Lock the palette so that TAD does not recreate new colors.
7. Set the offset of color zero to zero.
8. Set the number of colors to be used to the total number of colors available.
9. Rerender the image.
10. Save the image to disk.

The image saved to disk will incorporate all of the color available in the screen mode you've chosen including the several which you picked by hand. This technique allows you to mix automatically chosen colors with ones which you have manually chosen.

4.10 Merging Palettes

Another variation on the technique outlined above is the ability to merge the palettes of several pictures into a single palette which can be used to display several pictures on the same screen at the same time.

By systematically locking and freeing the palette and restricting the number of colors which can be chosen at any given time, you can extract key color information from several images to produce a palette tuned to the needs of all of the images as a group. Then, render each image using the entire palette to get even better results.

5 Summary

CDTV is represents break-through technology not because it contains whizz-bang Amiga technology, but because it bundles this technology in a package perfect for integration into the typical consumer's lifestyle.

While graphics may be important to computers... imagery is what's important to CDTV. Applications which exploit CDTV's rich imaging capability stand a significantly better chance at mass acceptance than those which do not.

Never forget, part of CDTV is quite literally TV. Mastering the techniques and addressing issues I have described will make the imagery in your CDTV applications more vivid and true to life and therefore, make them more readily accepted.

(

(

(



New Commodore Amiga Products

by Jeff Porter
Director, Product Development

WARNING: The information contained herein is for the sole purpose of informing Commodore's Third Party hardware and software developers of Commodore's new product plans. This information may not be published, copied, or distributed in any form, including electronic, without the prior written consent of Commodore Business Machines, Inc.

It is important for Commodore and the developer community to work together during the development of new hardware or software products because:

1. Commodore is able to get early feedback on new products.
2. Developers are better able to plan their product to compliment Commodore's products.
3. Enables Commodore and Developers to fully support new products and features at launch time.

We feel that Commodore is entering the most exciting chapter in our history since the original A1000 nearly four years ago. During the past few years, Commodore has been working on the key technology items that are required for the next generation of Amiga computers:

- ☐ **A2620/A2630** The first piece of the puzzle is bigger, faster and better microprocessor technology. This started with the A2620 - a 14MHz 68020 card for the A2000. But this was not good enough. The A2630 provided a full asynchronous design with a 25MHz 68030 on a single plug in card that included up to 4 megabytes of 32 bit fast RAM.
- ☐ **A590/A2091** The second piece of the puzzle is a fast and efficient custom DMA interface for SCSI hard drives. The A590 provides a 20MB add on for the A500 with the capability of adding up to 2 megabytes of fast RAM, in a compact, attractively styled external case. The A2091 makes use of the same circuitry in a plug in card form factor for the A2000. Thoroughly enhanced system software rounds out the technology which makes using a hard disk much easier than on any other platform.

- Commodore's strength has traditionally been in it's vertical integration, especially our in-house wafer fabrication facility in Valley Forge, PA. Over the past 18 months, we have added the capability to design and manufacture gate arrays in addition to full custom designs. A vast array of high technology CAD tools have helped us leverage our vertical integration once more to design more chips, more quickly, and for less cost than our competitors.

It is these key items that lay the foundation for the new Commodore Amiga 3000 Personal Computer. Perhaps Personal Workstation is more appropriate, since this small desktop design houses a true 32 bit architecture. At the heart of the Amiga 3000 is a 16MHz 68030/68881 or a 25MHz 68030/68882 providing the basis for a full 32-bit design all around (32-bit RAM, 32-bit ROM, 32-bit DMA, 32-bit expansion bus, and a 32-bit interface to Chip RAM).

A3000 Architecture

Inside, the A3000 has four horizontal expansion slots. The expansion bus features a new ZorroIII design, which is compatible with nearly all ZorroII cards, while providing additional address space and speed over Zorro II, yet maintaining the SAME physical plug in card connector. Two of the slots have PC-AT compatible connectors in line, and one of the four slots has the video slot in line with the 100 pin slot for future software controlled video cards.

For drives, the A3000 has three bays. The standard configuration is with a 40 megabyte 19ms hard drive, and one 3.5" floppy drive. One additional free bay is available for either a second floppy or second hard drive. All drives are easily installed or removed with just one screw. A 135-watt power supply powers the system with a convenient front mounted power switch.

On board, the A3000 features five new custom chips. Fat Buster controls the new Zorro III expansion bus. Ramsey is a fast RAM controller which supports up to 4MB of RAM using 256Kx4 DRAMs, or 16MB of RAM using 1Mx4 DRAMs. Super DMAC is a 32-bit DMA controller for the hard disk drive. Fat Gary is the system traffic cop for each of the system's resources. Amber is the heart of the display enhancer circuitry. Using the latest in video line and field memories, the Amber chip deinterlaces and scan doubles 15KHz video into 31KHz video. Of course, the real heart of any Amiga is Agnus which has been extended to address 2MB of chip RAM. The new ECS Denise chip has many new features which include Super-Hires mode, Productivity mode, and programmable transparency for genlocking. For more information on the Amiga 3000 architecture and design of plug-in-cards, be sure to stop by the technical sessions being given by Commodore hardware engineers Greg Berlin, Dave Haynie and Scott Scaeffler.

Externally the A3000 features a full compliment of connectors. In addition to the standard serial, parallel, and floppy connectors, the A3000 features a DB25 SCSI connector. Two video outputs are provided for simultaneous 15 and 31KHz video. Standard stereo RCA jacks round out the rear of the unit. The joystick and keyboard connectors are conveniently located on the side of the unit for easy access.

One additional slot is provided internally for complete access to the 68030 microprocessor bus. This slot has obvious uses for faster processors such as the 68040. The A3000 is complimented with the new Commodore 1950 multiscan monitor and A10 Stereo Speakers. The 1950 monitor features overscan and interlace compatibility unlike most multi-frequency monitors on the market today. It's .31mm dot pitch and 15 to 35 KHz design make this monitor the perfect match for the A3000.

The New System Software

It is important however to stress that application software must take advantage of these new features. Having the capability of programmable resolutions is a powerful feature not matched on the market today. Imagine if you will a spreadsheet or desktop publishing package, or even a schematic capture package that opens a 2000x2000 Workbench application which can scroll around both vertically and horizontally with the greatest of ease. The A3000 can do this today like no other computer in its price range and is begging to be utilized.

The new 2.0 release of the Amiga operating system Software has a wonderful new look and feel, with more user preferences than ever before. Now on a multitasking operating system we have had to be polite of other programs that are running, but have we always been polite to the *user*? Unfortunately, the answer is no. Be polite to your end user. Let *the user* determine the font they like, and the resolution they want, and other preferences they have specified.

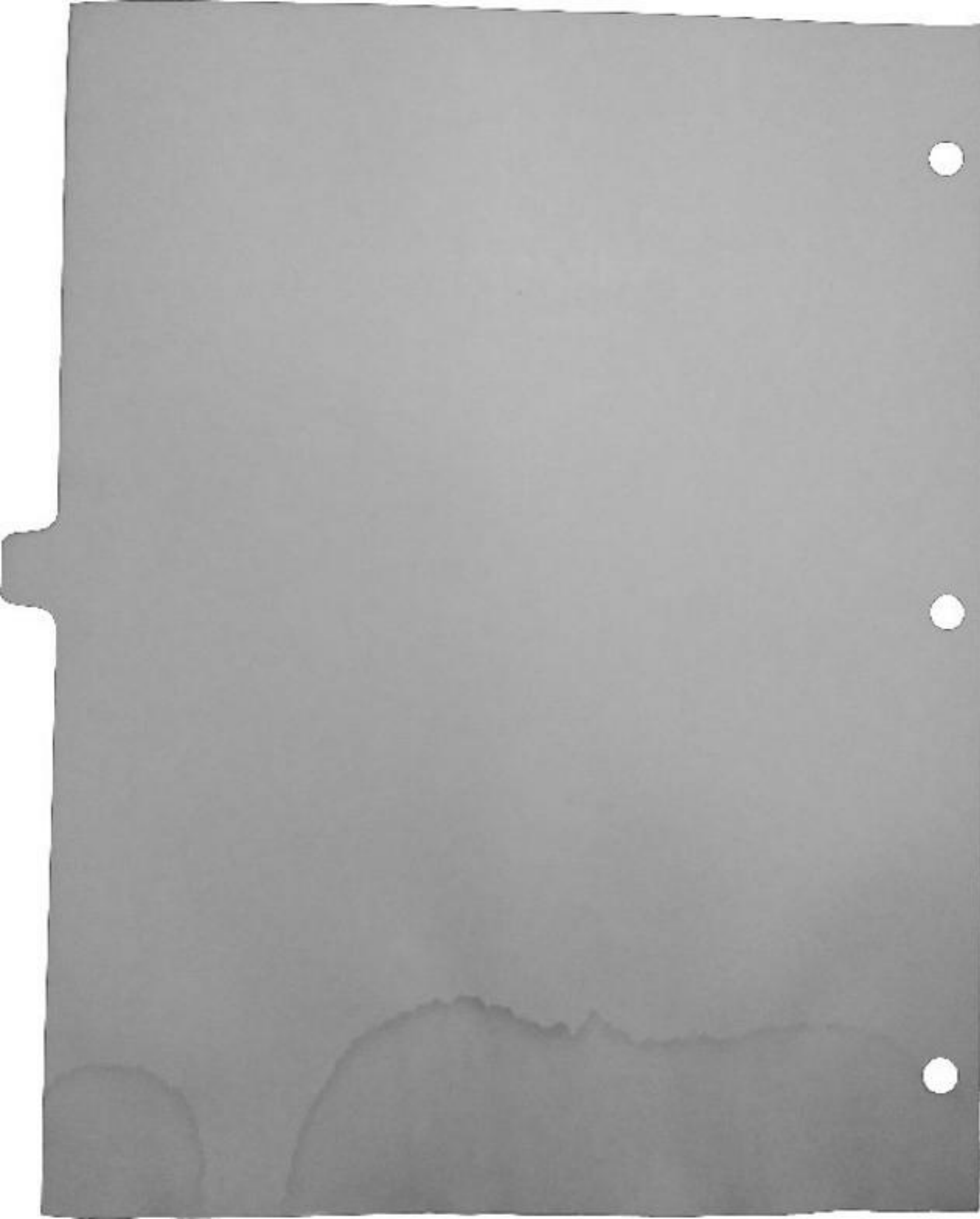
Allowing the end user to select the font of his choice is *not* the same as defaulting to TOPAZ8. We have some very nice Adobe Screen fonts: Times Roman, Helvetica, and Courier which for the first time, have the resolution on the A3000 to really make use of them. And with the addition of ECS chips to the A500 and A2000, under 2.0 productivity mode displays will look just as good as they do on an A3000, without the expense.

Networking and the Amiga

Finally, networking is absolutely critical to our future with the Amiga. With the recently introduced Arcnet and Ethernet cards, as well as the multiport serial card, the hardware is in place to connect Amigas in many different situations. Our direction for networking is also quite flexible. Dale Luck will be speaking about our new networking standards. Our goal is that applications should run over any physical layer (Ethernet, Arcnet, et al.) and any protocol (NFS, TCP-IP, Novell, et al.) without the application program needing to deal with it. After all, we *are* multitasking. Martin Hunt will be discussing our Amiga/NFS software, and Oxxi will be discussing our port of Novell Netware for the Amiga. Joe Augenbraun and Greg Rapp will be conducting our own "mini-connectathon" for Novell in the computer room. Please make a special effort to try your programs before the conference is out.

Also at the conference to discuss the A2410 High Resolution Color Graphics Card is Rich Miner and his team from the University of Lowell. Rich will be discussing both the hardware and software, and specifically TIGA on the Amiga. If you are writing high end applications that need 1024x768 non-interlaced with 256 colors out of 16 million - you won't want to miss this talk.

In summary, we've come a long way these past twelve months, and we need your support to make it all happen. We need you to exploit the new features of the A3000 and 2.0. We need your support for new standards and the new look. Let the user choose his own preferences for fonts and resolution, and listen to him. ♦



Commodore TCP/IP

by Martin Hunt

Brief Overview of TCP/IP

The Internet Protocol Suite was developed by the US Department of Defense to run on the ARPANET. It is often referred to as the TCP/IP protocol suite, after its two most common protocols, Transmission Control Protocol and Internet Protocol. To encourage its use, the DoD funded a version of TCP/IP for UNIX, which Berkeley included in their UNIX distribution. By 1983, all computers on the Internet were required to use TCP/IP.

TCP/IP is more than just a protocol. It is a whole family of protocols and applications. The following chart shows the 5 network layers and the corresponding protocols or applications.

Layer	Internet Protocol(s)
Application	Telnet, FTP, TFTP, SMTP, DNS
Transport	TCP, UDP
Network	IP, ICMP
Data Link	ARP, Ethernet Driver
Physical	Ethernet board and cable, RS232 port, etc.

Physical Layer

The physical layer (or hardware layer) is concerned simply with getting 1s and 0s from one location to another. Typical hardware appropriate for TCP/IP includes Ethernet, Arcnet, packet radio, satellite links, high-speed modems, etc. Unlike some protocols, TCP/IP deals easily with garbled or lost data, making it useful over almost any transmission media.

Data Link Layer

This layer is concerned with breaking the data into manageable chunks, usually called packets, and moving it from physical address to physical address. The Address Resolution Protocol (ARP) works at this level. ARP translates 48-bit physical addresses into 32-bit Internet addresses.

Network Layer

The network layer handles routing from one Internet address to another, across networks if necessary. There are two Internet network layer protocols: IP and ICMP. IP provides computer-to-computer communication. ICMP handles error and control messages.

Transport Layer

The transport layer handles communications between processes on different machines. There are two Internet transport protocols: TCP and UDP. TCP (Transmission Control Protocol) is a reliable, connection-oriented protocol. UDP (User Datagram Protocol) is connectionless. The protocol used depends on the application.

Application Layer

Some of the more common TCP/IP applications include:

- ☐ Telnet The Telnet protocol enables terminals to communicate with processes over networks.
- ☐ FTP and TFTP File Transfer Protocol (FTP) and Trivial File transfer Protocol (TFTP) transfer files to and from remote hosts.
- ☐ DNS The Domain Name Service protocol provides a name-to-address service.
- ☐ SMTP The Simple Mail Transfer Protocol provides electronic mail service.

UNIX Remote Services

UNIX includes many networking applications in addition to the standard TCP/IP suite applications. Like all network applications, both client and servers are required for a connection to succeed.

- ☐ rcp Provides remote copy service.
- ☐ rsh Provides for remote execution of commands.
- ☐ rlogin Remote login service that understands trusted hosts and has less overhead than Telnet.
- ☐ finger Allows you to find out about users on remote machines
- ☐ ping Determines if another computer is alive

Sun Extensions

- ☐ XDR The eXternal Data Representation is a standard for the description and encoding of data. It provides a common way to represent data over the network.
- ☐ RPC The Remote Procedure Call specification provides a procedure-oriented interface to remote services.
- ☐ NFS The Sun Network FileSystem protocol provides transparent remote access to shared files across networks. The NFS protocol is designed to be portable across different machines, operating systems, and network architectures. This portability is achieved through the use of RPC and XDR.

The NFS protocol was intended to be as stateless as possible. That is, a server should not need to maintain any information about any of its clients. Stateless servers have an advantage over other servers in the event of a failure. An NFS server that crashes need not worry about open files, locks, etc. It merely reboots and waits for the next request to process.

Commodore TCP/IP Software Features

The Commodore TCP/IP software supports the A2065 Ethernet card or Ameristar Ethernet cards. The Commodore AS225 software can support as many cards as you can connect to a machine (normally up to 5). It works on any Amiga under 1.3 or 2.0. Requires 1MB RAM.

The A2065 is a Zorro II card. It provides 15 pin AUI connector for use with thick Ethernet (10BASE5) and a BNC connector for use with thin Ethernet (10BASE2). It has been tested with Amiga 2000, 2500 and 3000s.

The Commodore TCP/IP software supports the following basic protocols:

- ☐ ARP ☐ ICMP
- ☐ IP ☐ UDP
- ☐ TCP

The following applications are included:

Telnet	(client only)
FTP	(client and server)
TFTP	(client and server)
rlogin	(client only)
rloginVT	(rlogin with VT52/100 emulation, client only)
ping	(client and server)
finger	(client and server)
rsh	(client and server, but one command at a time)
rcp	(client only)
route	(client and server)

The following commands and diagnostic programs are included:

rpcinfo	ls
showmount	chmod
lance-test	netstat
passwd	arp

Network FileSystem (NFS)

The networking software also supports Sun XDR, RPC, and NFS (client only). NFS client software gives you the ability to mount disks served by an NFS server. Once mounted, access to remote NFS volumes is completely transparent. That is, you access remote files just like they were on a local partition of your hard drive.

For example, I have an account on cbmvax in directory */usr/softeng/martin*. In my startup-sequence I execute:

```
nfsmgr mount cbmvax:/usr/softeng/martin martin:
```

This creates a remote partition *martin:* that is functionally equivalent to my local *work:* partition. If I'm running Workbench, an icon comes up for *martin:* and I can open it, move icons into it, etc. Unless I watch the hard disk light, I don't even realize that the files are being stored across the network, not locally.

NFS server software is available on most UNIX machines, as well as many other operating systems.

Usage Examples

ping

Ping is used to see if another machine is alive or to check to see if a connection exists to the other machine.

Usage: ping [-drv] host [data size] [npackets]

>ping cbmvax

```
PING cbmvax (192.9.210.4): 56 data bytes
64 bytes from 192.9.210.4: icmp_seq=0. time=16. ms
64 bytes from 192.9.210.4: icmp_seq=1. time=0. ms
64 bytes from 192.9.210.4: icmp_seq=2. time=0. ms
```

cbmvax PING Statistics:

3 packets transmitted, 3 packets received, 0% packet loss round-trip (ms)
min/avg/max = 0/5/16

finger

Finger gives information about remote users

Usage: finger [user][@host]

>finger @ghostwheel

```
[ghostwheel]
Login      Name           TTY Idle   When      Where
rsbx       Ray Brand      p0   9d Mon 16:04  toaster
martin     Martin Hunt    p1      Thu 12:08  pepsi
```

>finger martin@ghostwheel

```
[ghostwheel]
Login name: martin           In real life: Martin Hunt
Directory: /usr/ginger/martin Shell: /bin/tcsh
On since Jun 21 12:08:24 on ttypl from pepsi
47 seconds Idle Time
No unread mail
No Plan.
```

rcp

rcp is the UNIX remote copy command. In UNIX,

> rcp my_file host2:

would copy *my_file* to *host2* (in your home directory, by default)

Because the colon ":" is used for volume names on the Amiga, Amiga rcp uses an equal sign instead. So,

> rcp startup-sequence cbmvax=

would copy your startup-sequence to cbmvax. You can also use

> rcp startup-sequence cbmvax=start

to copy *startup-sequence* to *start* on cbmvax.

rsh

rsh executes remote commands. On the Amiga, rsh works just like UNIX except an rsh into an Amiga cannot start up a shell.

Usage: rsh host commands...

The following command execute the *status* command on the remote Amiga named *coke*:

```
> rsh coke status
Process 1: Loaded as command: inet:c/NFSc
Process 2: Loaded as command: dh0:bin/fixlace
Process 3: Loaded as command: dh0:bin/dlineart
Process 4: No command loaded.
Process 5: Loaded as command: inet:serv/portmapd
Process 6: Loaded as command: inet:serv/inetd
Process 7: No command loaded.
Process 9: Loaded as command: inet:serv/rshd
Process 10: Loaded as command: status
```

For another example of using rsh, see the sample script on the last page.

rpcinfo

Gives RPC information on a remote server

```
>rpcinfo -p ghostwheel
```

program	vers	proto	port	
100004	2	udp	1027	ypserv
100004	2	tcp	1024	ypserv
100004	1	udp	1027	ypserv
100004	1	tcp	1024	ypserv
100007	2	tcp	1025	ypbind
100007	2	udp	1035	ypbind
100007	1	tcp	1025	ypbind
100007	1	udp	1035	ypbind
100009	1	udp	1023	yppasswd
100003	2	udp	2049	nfs
100024	1	udp	1087	status
100024	1	tcp	1031	status
100021	1	tcp	1032	nlockmgr
100021	1	udp	1092	nlockmgr
100020	1	udp	1095	llockmgr
100020	1	tcp	1033	llockmgr
100021	2	tcp	1034	nlockmgr
100012	1	udp	1115	sprayd
100011	1	udp	1117	rquotad
100005	1	udp	1119	mountd
100008	1	udp	1121	walld
100002	1	udp	1123	rusersd
100002	2	udp	1123	rusersd
100001	1	udp	1126	rstat_svc
100001	2	udp	1126	rstat_svc
100001	3	udp	1126	rstat_svc
100015	6	udp	8769	selection_svc

showmount

Shows which remote volumes may be mounted.

```
>showmount ghostwheel
```

```
FilesystemGroups
/usr  heartofgold, allsun, allsoft,
/usr/ghostwheel heartofgold, allsun, allsoft,
/usr.MC68010/ghostwheel heartofgold, baby, allsun, allsoft,
/usr.MC68010 heartofgold, allsun, allsoft,
```

lance-test

Tests A2065 cards.

```
>lance-test diags
```

Ethernet address of board is 00:80:10:00:00:01

Ethernet Controller Diagnostics

```
Buffer memory test..... PASS
LANCE configuration test..... PASS
Interrupt test..... PASS
LANCE collision logic test..... PASS
Internal loopback test..... PASS
```

Controller passed diagnostics.

passwd

Updates the local password file. Used for remote access from FTP.

arp

Get internet to Ethernet address mappings.

```
>arp
```

```
usage: arp hostname
      arp -a
      arp -d hostname
      arp -s hostname ether_addr [temp] [pub] [trail]
      arp -f filename
```

```
>arp -a
```

```
cbmvax (192.9.210.4) at aa:0:4:0:14:8
ghostwheel (192.9.210.50) at 8:0:20:1:e:1f
```

netstat

Print network statistics.

usage: netstat [-Aaihmnrst] [-p proto] [-I interface]

>netstat -p tcp

tcp:

```
1619 packets sent
  699 data packets (1539 bytes)
  0 data packets (0 bytes) retransmitted
  865 ack-only packets (788 delayed)
  0 URG only packets
  0 window probe packets
  0 window update packets
  55 control packets
1257 packets received
  741 acks (for 1581 bytes)
  29 duplicate acks
  0 acks for unsent data
  1047 packets (115880 bytes) received in-sequence
  15 completely duplicate packets (15 bytes)
  0 packets with some dup. data (0 bytes duped)
  15 out-of-order packets (0 bytes)
  0 packets (0 bytes) of data after window
  0 window probes
  5 window update packets
  0 packets received after close
  0 discarded for bad checksums
  0 discarded for bad header offset fields
  0 discarded because packet too short
22 connection requests
11 connection accepts
31 connections established (including accepts)
44 connections closed (including 0 drops)
2 embryonic connections dropped
741 segments updated rtt (of 763 attempts)
2 retransmit timeouts
  0 connections dropped by rexmit timeout
0 persist timeouts
0 keepalive timeouts
  0 keepalive probes sent
  0 connections dropped by keepalive
```

>netstat -I ae0

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
ae0	1500	cbm	pepsi	8390	0	2378	0	0

ls

UNIX ls command for the Amiga. Shows owners and protection bits on NFS volumes.

For example,

```
>list martin:tmp
```

```
Directory "martin:tmp" on Wednesday 20-Jun-90
.info          81 ----rwed Thursday 11:23:16
readme        7128 ----rwed Friday 21:02:37
rfc-index     136342 ----rw-d Friday 14:48:33
work          Dir --p-rwed Today 15:31:06
3 files - 1 directory - 308 blocks used
```

```
>ls -l martin:tmp
```

```
drwxrwxrwx 406    14    90-06-20 15:31:06    0    Dir work
-rwxrw-rwx 406    14    90-06-14 11:23:16    2    81 .info
-rwxrw-rwx 406    14    90-06-15 21:02:37   14    7128 readme
-rw-rw-r-- 406    14    90-06-15 14:48:33  288   136342 rfc-index
Dirs:1    Files:3    Blocks:304    Bytes:143551
```

chmod

UNIX chmod function for the Amiga. Can modify NFS volume protection bits.

```
>chmod a+w martin:tmp/rfc-index
```

```
>ls -l martin:tmp
```

```
drwxrwxrwx 406    14    90-06-20 15:31:06    0    Dir work
-rwxrw-rwx 406    14    90-06-14 11:23:16    2    81 .info
-rwxrw-rwx 406    14    90-06-15 21:02:37   14    7128 readme
-rw-rw-rw- 406    14    90-06-15 14:48:33  288   136342 rfc-index
Dirs:1    Files:3    Blocks:304    Bytes:143551
```

SANA

SANA (Standard Amiga Network Architecture) is the future standard for Amiga networking. The current (Beta) release of the networking software does not yet support SANA. It will probably be released almost unchanged as version 1.0. However, conversion to SANA will be our highest priority. All future networking releases will be fully SANA compliant.

Once Commodore's SANA implementation is complete, we will make available complete specifications, sample code, and libraries for linking to the socket library.

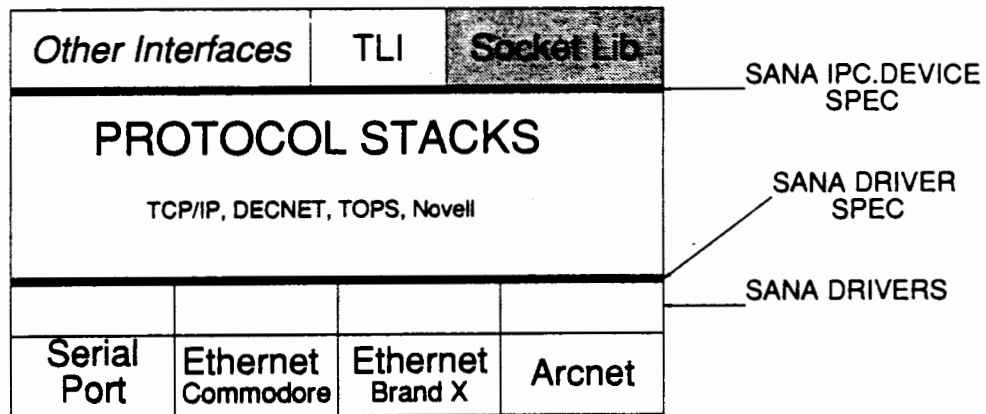
We strongly encourage anyone developing networking products for the Amiga to use SANA. Software conforming to SANA will be compatible with all other SANA-compliant software. Additionally, all SANA software will be able to use any hardware with a SANA driver. This will make it possible for companies to write networking software that will automatically be compatible with all the networking cards available. It will also make it possible for someone to develop a new network card (for example, FDDI) without worrying about which applications will be able to use the card.

Future Improvements

In addition to the upgrading to SANA, we are considering many improvements to the networking software. Currently it is very UNIX-like and doesn't fit in well with the Amiga philosophy. That will certainly be improved. There are also many applications that we are considering, including an NFS server, SMTP, SLIP, and support for our Arcnet boards. Of course, once we release a SANA version of the networking software, anyone can write networking applications.

Figure 1: The Components of SANA

Standard Amiga Network Architecture



Security

Currently the Amiga networking software, like many PC-based networking products, poses a potential security problem. Every PC is configured with a machine name and internet number. Every user has a username, user ID (UID), and group ID (GID). The problem is that these are not secure and may be easily changed by a knowledgeable user.

Some of the important security files you should be aware of are:

hosts.equiv

This is a system file that lists the trusted hosts. If a machine is included in this list, then rlogin, rsh, rcp, etc. will be permitted freely from that machine to your machine.

exports

This system file lists which directories will be exported via NFS and which machines may access them.

.rhosts

This is a user file in the user's home directory. It permits rlogins from specific hosts without prompting for a password.

.netrc

This is a user file that lists the name and passwords to automatically use when FTP'ing to a remote machine. This is always a security problem because account names and passwords are listed here in clear text.

The interim solution to the security problem is to tell your users to not use *.netrc* and *.rhosts* files. You should also make sure that the *hosts.equiv* file contains no unsecure machines.

Preventing illegal access via NFS is difficult, because PCs can change their names easily. You can make this more difficult for potential troublemakers by making permanent arp entries for each PC Ethernet address. Of course this will make things more difficult in case of Ethernet board trouble, but in some cases the additional security may be worth it.

The best solution to the security problem is to use some kind of authentication service. This means that before you use your PC on the network, you will first have to provide a valid username and password to some central server which maintains a secure master password file. We are looking at several different methods to do this.

Sample Script for Remote Printing

This script assumes the use of NFS. It could be written to use rcp if NFS is not available.

```
. rprint
.key file/a,homevol/k,username/k,printer/k,number/k,sides/k,type/k
.bra {
.ket }
.def number 2
.def sides 1
.def type "ascii"
.def printer "lpl"
.def homevol "martin:"
.def username "martin"

IF NOT EXISTS {homevol}.prspool
mkdir {homevol}.prspool
ENDIF

delete >NIL: {homevol}.prspool/#?

copy {file} {homevol}.prspool

run rsh cbmvax -l {username} "cd ~{username}/.prspool;lpr -P{printer}
-N{number} -K{sides} -D{type} *"

echo "Done printing"
```





**Discussion of
Amiga Client NetWare
for
Novell NetWare**

Developed by:

Scott A. Martin
Timothy S. Patrick
Michael A. Uman

Beta-Testing:

Greg Rapp

Documentation:

Patricia Cummings

Concept and Administration:

John Houston

(C) 1990 Oxxi, Inc.

Table of Contents

Overview	
NetWare Advantages	4
Amiga Client NetWare Requirements	7
NetWare Market Data	8
Record-Locking Using Amiga Client NetWare With WorkBench 1.3	
Function List	10
Novell NetWare Routines in Amiga Client NetWare	
Summary List	14
Internet Packet Exchange	
IPX Device	18
IPX Packet Structure	19
Event Control Block (ECB)	21
Event Service Routine (ESR)	23
Summary of IPX Services	24
NetWare Core Protocol Library	
Definition	28
Explanation of Services	29
NCP Library Function Calls, by category	31
Glossary	41

Amiga Client NetWare

Overview

(C) 1990 Oxxi, Inc.

NetWare Advantages

Amiga Client NetWare adapts any standard Novell NetWare network to allow Amiga computers to act as network clients. Amiga NetWare software installs quickly on an existing Novell NetWare system running Version 2.15 or higher, takes only minutes, and requires no prior preparation. Once Amiga NetWare is installed, Amiga users on the network enjoy all the NetWare capabilities available to PC and Macintosh workstations. Amiga workstations retain their multi-tasking, graphic environment in addition to the full range of Novell NetWare functions. Users receive their own personal set of Amiga Preferences -- including network printing preferences -- automatically when they login from any Amiga on the network.

Full NetWare Functionality for the Amiga WorkStation

Novell-style utilities provide easy selection of standard Novell NetWare functions on the Amiga, including:

- File Management and Backup/Archiving
- Printer Definition and Print Job Control
- Inter-User Communication via Mail and Messaging
- Full Accounting Records for all NetWare Services
- User Access and Group Account Management

Centralized Resources

Amiga workstation users benefit from centralized software and data resources. They can update or change programs once, on the server, rather than for each workstation individually; change data and information globally to affect every network user; and hide or give access to resources on an individual or group basis. Vital files are kept on the server, not on vulnerable floppy disks.

Individual Preferences

Individual user preferences are supplied automatically at login from any workstation. Login scripts can execute both Novell NetWare script commands and AmigaDOS commands, to allow both system-wide and individual customized logins. Network or local printing is selectable from a NetWare-specific Preferences tool.

Data Integrity

Amiga Client NetWare for Novell NetWare includes record-locking and file-locking functions for protection of shared data files on the file server. These safeguards allow multiple users to access the same data concurrently. Protection at the record level is not available for files stored on the local

Amiga workstation under WorkBench 1.3, but file-locking for locally-stored files is provided under the Amiga WorkBench Version 1.3 and higher. WorkBench 2.0 will have the record-locking feature as well.

Inter-Platform Communication

The Amiga workstation shares resources and data with other workstations, regardless of platform. Amiga Client NetWare provides access to and full file-transfer utility between the Amiga and IBM PC and Macintosh platforms, plus the full range of other NetWare-supported platforms, via the network file server. Information is transferred quickly and easily over the network, even to remote locations.

Shared Peripherals

Printers and other peripherals shared on a network are more fully utilized, reducing idle time. The Amiga workstation can print from a file or directly from the Amiga application, using the NetWare server as an intelligent spooler device. Printing to a network printer via the Novell NetWare system is 100% transparent to the Amiga application.

Shared Applications

Amiga NetWare users will have access to complex shared-use applications such as Electronic Mail systems and multi-user accounting systems. Network versions of programs provide shared access and use-management for multiple users. Inexpensive workstations can use Novell NetWare file server in place of multiple local hard disks. Sophisticated caching algorithms allow server processing to approach the speed of a local hard drive.

Send Messages

A quick-send utility built into Novell NetWare allows private communication with any user, or group- and network-wide messaging with a single command. Amigas have the same peer-to-peer abilities as PCs on the Novell NetWare network. The user can also communicate via bridges to other networks and mainframes, allowing file-transfer, electronic messaging and access through third-party bridges from Novell NetWare network. Connect your Amiga workstation to IBM, Sun, DEC, and other network and mainframe platforms.

Security for Information, Applications

Individual user files can be protected by password and assignment of access rights. Confidential mail areas are automatically created for each user at the time the user's name is placed on the network. System security allows use of information, applications, and other resources by individuals or groups to be controlled. Network security can be designed to protect data

and software from theft or access by unauthorized persons. Novell NetWare allows assignment of multiple levels of access to files and directories, and shared peripherals.

Amiga Client NetWare Requirements

Software

Novell NetWare Version 2.15 or higher
 Installed* and operating on network file server
 (Currently will not work with NetWare 386 Version 3.0)

Amiga NetWare Version 2.15 or higher

Amiga WorkBench Version 1.3 or higher

Hardware

Amiga Workstation with 512 K or more

File Server

LAN Communications Card (CBM ARC-Net or Ethernet)
 (1) for each Amiga workstation
 (1) for file server

Cabling and other network connection hardware

Optional

Peripherals such as printers and plotters

Third-party additions to Novell NetWare such as bridges

- * Correct installation of the Novell NetWare network is critical to the operation of the Amiga Client NetWare software, and the installation process is not a trivial task; certified Novell dealers (distributors and installers) have completed several courses in this procedure. The initial network file server installation should be done by a certified Novell dealer. Novell makes classes in NetWare installation and maintenance available world-wide, either direct or through their distributors. It is highly recommended that file server installation be accomplished by someone who has attended one of these classes.

Novell NetWare Market Data

Over 500,000 installed servers worldwide

1989 NetWare User Group survey = 1.8 servers per network
Independent research report = 1.8 servers per network

6,000,000 clients

1989 NetWare User Group survey = 11.7 clients per server
Independent research report = 13.3 clients per server

Largest body of applications of any network operating system

Over 5000 applications
Over 1500 application developers (ISV's)

Largest market share of all network operating systems

40-70% market forecasts by industry analyst
Nearest competitor = 14-17%

Widest desktop operating support

Amiga
MS-DOS
Apple Macintosh
OS/2
Microsoft Windows
Desqview

Broadest server platform support

PC's (8, 16 and 32-bit)
UNIX (Prime, NCR, Data General, Altos, Interactive, SCO, Integraph, Northern Telecom, Unisys, Zenith Data Systems, Pyramid, Hewlett Packard, etc.)
Other operating systems (DEC-VMS, Data General-AOS/VS, Wang-VS, etc.)
Super servers (NetFRAME, Compaq-System-Pro, etc.)

**Record-Locking Using
Amiga Client NetWare
With WorkBench 1.3**

(C) 1990 Oxxi, Inc.

Record-Locking Using Amiga Client NetWare With WorkBench 1.3

Four record-locking routines are available on the Amiga NetWare system. These routines, which will be incorporated into AmigaDOS 2.0 to provide record-locking capabilities at the individual Amiga computer, are available under WorkBench 1.3 *if the application stores and handles these records on the Novell NetWare file server.*

Amiga NetWare Record-Locking Routines:

NAME

LockRecord -- Locks a portion of a file

SYNOPSIS

```
success = LockRecord (fh, offset, length, mode, timeout)
                D0          D1  D2   D3      D4    D5
```

ULONG LockRecord (BPTR, ULONG, ULONG, ULONG, ULONG)

FUNCTION

This locks a portion of a file for exclusive access. Timeout is how long to wait in ticks (1/50 second) for the record to be available.

Valid modes are:

```
RLOCK_EXCLUSIVE
RLOCK_EXCLUSIVE_IMMEDIATE
RLOCK_SHARED
RLOCK_SHARED_IMMEDIATE
```

For the immediate modes, timeout is ignored.

INPUTS

```
fh      - File handle for which to lock the record
offset  - Record start position
length  - Length of record in bytes
mode    - Type of lock requester
timeout- Timeout interval; 0 is legal
```

RESULT

success - Success or failure

NAME

LockRecords -- Lock a series of records

SYNOPSIS

success = LockRecords (record_array, timeout)
D0 D1 D2

BOOL LockRecords (struct RecLock *, ULONG)

FUNCTION

This locks several records within a file for exclusive access. Timeout is how long to wait in seconds for the records to be available.

INPUTS

record_array - List of records to be locked
timeout - Timeout interval; 0 is legal

RESULT

success - Success or failure

NAME

UnLockRecord -- Unlock a record

SYNOPSIS

success = UnLockRecord (fh, offset, length)
D0 D1 D2 D3

BOOL UnLockRecord (BPTR, ULONG, ULONG)

FUNCTION

This releases the specified lock on a file. Note that you must use the same filehandle you used to lock the record, and offset and length must be the same values used to lock it. Every LockRecord call must be balanced with an UnLockRecord call.

INPUTS

fh - File handle of locked record
offset - Record start position
length - Length of record in bytes

RESULT

success - Success or failure

NAME

UnLockRecords -- Unlock a list of records

SYNOPSIS

```
success = UnLockRecords (record_array)
D0                                     D1
```

```
BOOL UnLockRecords (struct RecLock *)
```

FUNCTION

This releases an array of record locks obtained using LockRecords. You should NOT modify the record array while you have the records locked. Every LockRecords call must be balanced with an UnLockRecord call.

INPUTS

record_array - List of records to be unlocked

RESULT

success - Success or failure

Novell NetWare Routines
in
Amiga Client NetWare

Summary List

(C) 1990 Oxxi, Inc.

Summary List of Novell NetWare Routines in Amiga Client NetWare

Utilities (also referred to as routines) in Novell NetWare are of several different types, depending on the type of station from which they can be called, and the way in which they operate.

Menu Utilities (Menu)

The Novell Menu Utilities are programs which allow the selection of many related NetWare functions in a custom window environment. These custom windows have been designed to emulate the way in which the Menu Utilities behave on the IBM PC workstation. This makes it easy to transfer Novell NetWare experience from the PC workstation to the Amiga client.

Command Line Utilities (CLU)

CLU routines are Novell NetWare command line utilities; they generally are called from the Amiga CLI/Shell, and do not, for the most part, have icons. The single tasks they perform are frequently also supported by selections from a Novell Menu Utility Program.

Supervisor

The Novell NetWare Supervisor is a kind of super-user, with access to server management functions not required by the day-to-day use of the network.

Utility/Routine	Type	Function
AMIGABACK	Menu	Back up and restore Amiga, Macintosh and MS-DOS format files from the file server to/from network or local destinations
ATOTAL	Supervisor	Call summary of PAUDIT file. (Only if accounting is installed)
ATTACH	CLU	Attach to other servers
BINDFIX	Supervisor	Repairs the bindery files
BINDREST	Supervisor	Restores previous version of bindery files
FCONSOLE	Menu	Virtual console utility
FILER	Menu	File and subdirectory info and management functions
FLAG	CLU	Change 4 of the 8 attributes of a set of files (read/write, shared, indexed, transactional)

Utility/Routine	Type	Function
GRANT	CLU	Grant trustee rights to users or groups
HIDEFILE	Supervisor	Hide files so they can't be listed
LOGIN	CLU	Initiate session with network, run login script, attach
LOGOUT	CLU	End session with network, detach from all attached servers or from specified servers
MAKEUSER	Menu	Creates USR files which can be used to create uniform users
MAKEUSER	Supervisor-CLU	Processes USR files outside of the MAKEUSER menu utility
MAP	CLU	View, add to, or delete drive mappings
NDIR	CLU	View file and subdirectory information
NPRINT	CLU	Send DOS or text files to a network printer
NVER	CLU	View the version of Netware
PAUDIT	Supervisor	Use to view system accounting records
PCONSOLE	Menu	Print Queue/Server Management Functions
PRINTCON	Menu	Print Job Configuration job definition functions
PRINTDEF	Menu	Define print devices and forms
PSTAT	CLU	View printer status
PURGE	CLU	Render all erased files unrecoverable
REMOVE	CLU	Remove a user or group from the trustee list of a given directory
REVOKE	CLU	Remove specific trustee rights from user or group
RIGHTS	CLU	View effective rights in given directory
SALVAGE	CLU	Recover files that have been ERASEd but not PURGEd

Utility/Routine	Type	Function
SECURITY	Supervisor	Checks for security "holes" in user list
SEND	CLU	Send a short message to user or group on network
SETPASS	CLU	Set or change your password on given server
SHOWFILE	Supervisor	Reveal files hidden by the HIDEFILE utility
SLIST	CLU	View list of servers
SYSCON	Menu	User, Group, Supervisor, and Accounting functions
SYSTIME	CLU	View day of week, time and date on server
TLIST	CLU	View the trustee list for a given directory
USERLIST	CLU	View list of current users on given server, with other info
VOLINFO	Menu	Displays Volume utilization information
WHOAMI	CLU	View current username, server, login date and time, etc. for each server to which attached

Internet Packet Exchange Protocol
Peer-to-Peer Communications for the
Amiga NetWare Network

(C) 1990 Oxxi, Inc.

IPX Device

IPX (Internet Packet Exchange), also known as peer-to-peer communications, is a protocol by which two workstations talk to each other with minimal file server intervention. In IPX, packets are sent and received on the Network Layer of the OSI (Open System Interconnection) model. The only involvement of the file server at this level is to deliver the packet to the correct workstation.

The OSI model contains seven layers used in peer-to-peer communications

Layer 1 - Physical Layer: consists of the boards, chips, cables and other equipment used in the communications.

Layer 2 - Data Link: concerns itself with tokens, bit pattern, and error detection.

Layer 3 - Network Layer: delivers communication packets on the network.

Layer 4 - Transport Layer: a watchdog used to make sure that all packets arrive in the proper sequence with no duplication.

Layer 5 - Session Layer: turns connections between machines on and off and sets up names and addresses for machines.

Layer 6 - Presentation Layer: translates data into understandable units for the local machine.

Layer 7 - Application Layer: sets up the interface between the network and an application used on the computer.

Because the IPX packets travel below the Transport layer, much of the error checking (i.e., packet tracing and ordering) is not done. When a packet is sent out, the machine does not automatically reply to the packet. It is up to each application to set up a information exchange system. Therefore, much care must be taken when using this protocol. There is no guarantee that a packet sent from a workstation will arrive at its destination. Because there is no sequence information, there is also no reason why two packets will not arrive in the reverse order that they were sent.

IPX does have the advantage of speed and performance because of the low overhead associated with it. IPX maintains a 95 percent success rate for accurate and ordered packet transfer.

IPX Packet Structure

Offset	Content	Type
0	Checksum	BYTE[2]
2	Length	BYTE[2]
4	Transport Control	BYTE
5	Packet Type	BYTE
6	Destination Network	BYTE[4]
10	Destination Node	BYTE[6]
16	Destination Socket	BYTE[2]
18	Source Network	BYTE[4]
22	Source Node	BYTE[6]
28	Source Socket	BYTE[2]
30	Data Portion	BYTE[0 to 546]

Checksum Field

This field is included to conform to the Xerox packet header definition. In IPX, the Checksum field is set to 0xFFFF.

Length Field

This contains the entire length of the packet, including 30 bytes for the header and up to 546 bytes for the data.

Transport Control

The Transport Control field is used by NetWare internetwork bridges. It is set to zero by IPX before the packet is sent.

Packet Type

This field indicates the type of service needed by the packet. IPX users should set this field to 0 (Unknown Packet Type) or 4 (Packet Exchange Packet).

Destination Network

This is a four byte network number assigned by the system administrator of a network.

Destination Node

The Destination Node field is six bytes containing the physical address of the destination node. A node address of FF FF FF FF FF FF broadcasts the packet to all nodes on the destination network.

Destination Socket

Sockets are used to direct a packet to the proper process at a workstation. Many sockets have been pre-determined or pre-defined, therefore, you must take care in choosing a socket number to use. Xerox has reserved the following socket numbers:

1	Routing Information Packet
2	Echo Protocol Packet
3	Error Handler Packet
20h-3Fh	Experimental
1h-BB8h	Registered with Xerox
BB9h-	Dynamically assignable

Xerox has assigned Novell a set of sockets for use by NetWare:

451	File Service Packet
452	Service Advertising Packet
453	Routing Information Packet
455	NetBIOS Packet
456	Diagnostic Packet

Those who are writing applications to be used on NetWare IPX should contact Novell and obtain a socket number (registered Novell sockets start at 8000h). Novell dynamic socket numbers begin at 4000h.

Source Network

Source Node

Source Socket

These three fields are the same as the destination fields except that they pertain to the sending station's information. IPX automatically sets these fields when a packet is sent.

Event Control Block (ECB)

An Event Control Block is a data structure containing information used to send and receive IPX packets. There are two types of ECB: Send ECB and Receive ECB. These ECB's are the same except the Receive ECB does not require a destination address in its Immediate Address field (see below), while the Send ECB does require the destination address.

Event Control Block Structure

The ECB structure is in two parts: the first is a 36-byte fixed portion and the second is a list of data fragments (see below).

Offset	Content	Type
0	Link Address	BYTE[4]
4	ESR Address	BYTE[4]
8	In Use Flag	BYTE
9	Completion Code	BYTE
10	Socket Number	WORD
12	IPX Workspace	BYTE[4]
16	Driver Workspace	BYTE[12]
28	Immediate Address	BYTE[4]
34	Fragment Count	WORD
36	Fragment Address 1	BYTE[4]
40	Fragment Size 1	BYTE[2]
42	Fragment Address 2	BYTE[4]
46	Fragment Size 2	BYTE[2]

(more fragments...)

Link Address

The Link Address field is used by IPX while the ECB is in use. It is used for keeping the ECB in a free list.

ESR (Event Service Routine) Address

This field contains the address of an application-defined routine that IPX calls when it sends (or receives, depending on the ECB) a packet. This routine will be described later in this paper. If no ESR is to be used, this field should be set to NULL.

In Use Flag

IPX uses this field to show the current status of the ECB. If this field is set to zero, then IPX is not currently using this ECB. Otherwise, IPX is actively using the ECB, or waiting for the occurrence of an event which the ECB will use.

Socket Number

This field contains the socket number on which the ECB will send (or has received) a packet.

IPX Workspace

This is a reserved field used by IPX, and should not be modified by the application, unless the ECB-In Use Flag is set to zero.

Driver Workspace

The Driver Workspace is a reserved field used by the network driver, and should not be modified by the application, unless the ECB-In Use Flag is set to zero.

Immediate Address

This field contains the address of the node to which a packet is to be sent, or from which it was received. If the packet did not arrive from a local network node, this field contains the address of the internetwork bridge used.

Fragment Count

This contains a count of the number of Fragment Descriptors (see below) which are associated with this ECB. There must always be at least one Fragment Descriptor with an ECB.

Fragment Descriptor

A Fragment Descriptor is made up of two parts:

Offset	Content	Type
0	Fragment Address	BYTE[4]
4	Fragment Size	BYTE[2]

The Fragment Address contains an address in memory to a data block which contains a packet to be sent, or which is ready to receive packet data. The Fragment Size field gives the size of this block of data. There must be at least one Fragment Descriptor with each ECB, and a count of how many there are should be tallied in the Fragment Count field.

In IPX, the first Fragment Descriptor entry must have a Fragment Size of at least 30 bytes (for the header). Also, all of the fragment sizes added together may not exceed the maximum packet size of 576 bytes.

Event Service Routine (ESR)

An Event Service Routine is an application-defined procedure which IPX calls after a certain event has occurred. An event can be one of the following: 1) a packet is sent; 2) a packet is received; 3) an event that rescheduled itself; or 4) an application-defined event. ESR's are called once the In Use Flag in the Event Control Block has been set to zero. An example of an ESR is a procedure that queues up incoming packets for the application to process.

Summary of IPX Services

IPXCancelEvent

This function cancels an ECB event. An example of an ECB event would be a send, listen, special purpose event, or an IPX scheduled or re-scheduled event. If an event is transferring data from or to the Fragment data areas, then the event cannot be cancelled. Also, because of differences in the way that Network hardware works, it may not be possible to successfully cancel an IPX send request.

IPXCheckReceive

This function check whether or not a packet is available to be received.

IPXCheckSocket

This function checks whether or not the specified socket is open.

IPXCloseSocket

This function closes an IPX socket. It also cancels any events that have been defined by the ECB's associated with the socket. There is no error if the socket was not already open. An application must close all sockets that it has opened before it destroys its Event Service Routines, otherwise the workstation could halt. This function should not be called from within an ESR.

IPXConnect

This function creates a virtual connection between the requesting workstation and the address specified in the Event Control Block.

IPXDisconnectFromTarget

This function disconnects all connections between the requesting workstation and the specified target workstation.

IPXGetDataAddress

This function returns the address of a specified data element. (Note: On the Amiga, this function returns the address sent to it. It is included only for compatibility).

IPXGetInternetworkAddress

Given a 4-byte network address and a 6-byte node address, this function returns a 12-byte internetwork address.

IPXGetIntervalMarker

This function returns an interval marker representing one IBM PC clock tick (approx. 1/18th second). An application can use this function to measure the elapsed time between two events. This timer is not intended for use with large time intervals, i.e. those in which the clock wraps to zero more than once between two events.

IPXGetLocalTarget

This function returns the value that will be put in an Event Control Block's Immediate Address field. The function is passed a 12-byte field consisting of the following:

Bytes 0-3	= Network Number
Bytes 4-9	= Node Number
Bytes 10-11	= Socket Number

Returned is a six byte Immediate Address and the approximate time that it will take to send a packet to the target.

IPXGetStatistics

This function returns diagnostic statistics maintained by IPX.

IPXGetVersion

This function returns the major and minor version and the revision of the IPX driver that is installed at the workstation.

IPXHoldEvent

This function temporarily halts an ECB which has already been submitted. The ECB can be restarted with the IPXServiceEvent function.

IPXInitialize

This function gets the entry address for the IPX interface. This function must be called before any other IPX function can be used.

IPXListenForPacket

This function tells IPX to wait for a packet. The address of an Event Control Block is passed to this function, which in turn points to the Fragment Address(es) and (optionally) an Event Service Routine. The socket specified in the ECB must already be open. There may be any number of ECB's listening for packets.

IPXOpenSocket

This function opens an IPX socket. This must be done before a packet can be received on that socket. If a socket number of 0x0000 is sent to the function, the first available socket in the range 0x4000 to 0x5000 will be opened. This is a dynamic socket as opposed to an assigned socket. A socket may not be opened twice. The number of open sockets that a workstation may have ranges from the default of 20 to a maximum of 150. This value is configurable.

IPXRelinquishControl

This function is used on workstations that are co-resident with a NetWare file server or bridge. The function temporarily suspends the application so that the file server or bridge can process its information.

IPXResetStatistics

This function allows certain IPX statistical values to be reset by the application.

IPXScheduleIPXEvent

This function schedules an IPX event to occur after a specified time interval expires. The interval may be between 0 and 65,535 clock ticks. If this function is called twice with the same ECB, then its timer is reset and begins to count down again. The ECB given to this routine specifies a send or a receive IPX request and (optionally) an Event Service Routine. This function must never be passed an ECB that is currently in use by IPX.

IPXScheduleSpecialEvent

This function sets up an events using an Event Control Block that will engage after a specific time interval expires. The interval may be between 0 and 65,535 clock ticks. If this function is called twice with the same ECB, then its timer is reset and begins to count down again.

IPXSendPacket

This function sends an IPX packet to the destination specified in the Event Control Block. The specified socket need NOT be open to send the packet. The ECB contains the Fragment Descriptors where the packet is stored in memory and (optionally) it contains the address of an Event Service Routine.

IPXServiceEvent

This function restarts an ECB which was halted with the IPXHoldEvent function.

Overview of Netware Core Protocol Library

(C) 1990 Oxxi, Inc.

NetWare Core Protocol (NCP) Library

The **NetWare Core Protocol (NCP)** library is the library utilized by all programs which require communication with the Novell system. These calls are broken into several categories describing their function:

- Accounting Services
- AFP Services
- Bindery Services
- Connection & Workstation Services
- Directory Services
- File Services
- Message Services
- Queue Services
- Synchronization Services
- Transaction Tracking Services

The NetWare Core Protocol library exists as a run-time Amiga Library called NCP.library. The current revision of the NCP library is 34.13.

Oxxi, Inc. will be producing a developer's package with complete information for using the NCP.library in developing software to take advantage of this access to the Novell NetWare file server for the Amiga client.

Explanation of Services

Accounting Services

Accounting services is an application program interface (API) that allows a server to charge the user for the use of its services.

Each value-added server determines its own charging rates for each type of service, and the file server bindery stores the list of authorized accounting servers and each client's accounting information.

AFP Services

By adhering strictly to Apple Computer's AppleTalk Filing Protocol (AFP), Netware allows DOS, Amiga and Apple data files to be stored on a Netware file server. Amiga and Apple files appear in DOS directory listings as DOS files, and in Amiga file lists with the complete Amiga filenames, while appearing in Apple folders as normal Apple file icons. Amiga icons will appear in Amiga disk and drawer windows as Amiga icons. AmigaDos utilizes the AFP services in order to overcome DOS's filename limitations.

Bindery Services

In a local area network environment, there is the need for a name service which provides a way for network resources and clients to be identified. A *resource* is anything that provides a service such as a file server, print server, or database server. A *client* is the user of the services provided by a resource. Each Netware file server maintains a database of the resources and clients available on the network. This special-purpose database is called the *bindery*.

Connection & Workstation Services

Connection services are used to create connections between workstations and file servers. Workstation services are primarily concerned with the internal tables that the shell uses to maintain these connections.

Directory Services

The Directory Services calls enable an application program to obtain information about volumes and directories; create, rename, and destroy directories; modify a directory's maximum rights mask; add and delete directory trustees; allocate and deallocate directory handles; and more.

File Services

Workstation operating systems provide functions which enable applications to open, read, write, close, and delete files. The NetWare shell allows these functions to be used to perform the same tasks on the NetWare (file server) files.

NetWare file services calls provide a set of supplementary functions that enable applications to manipulate extended file attributes, restore erased files, permanently delete files, set and scan file information, and copy files between directories on the same file server.

Message Services

Message Services is a set of NetWare APIs that enable applications to send broadcast messages *and pipe messages* to up to 100 specified target connections (workstations). The sending workstation and the target workstation must be attached to the same file server.

Queue Services

Queue Services make NetWare's Queue Management System (QMS) available to developers. QMS provides a central storage and queueing mechanism that enables users to create jobs that are added to the queue. These jobs can be services called remotely by an application at another node on the network.

A job, as used in the context of QMS, is an individual entry in a QMS queue. A job server (or value-added server) is an application that runs on a workstation, or as a Value-Added Process (VAP) on a file server or bridge.

Synchronization Services

Synchronization Services calls allow the application to control file access synchronization through file and record locking mechanisms.

Transaction Tracking Services

NetWare Transaction Tracking Services (TTS) is a feature that ensures data integrity on files that otherwise would be corrupted when updates on the files are interrupted by such things as hardware failures or power outages. A *transaction* is defined as a set of one or more operations that must be completed together to maintain file and database integrity. TTS guarantees that all writes within a transaction will be completed or none will be completed.

NCP Library Function Calls, by category

Accounting Functions:

GetAccountStatus():

Returns the account status of a bindery object.

SubmitAccountCharge():

Updates the account of a bindery object.

SubmitAccountHold():

Reserves a specified amount of an object's account pending a SubmitAccountCharge call.

SubmitAccountNote():

Adds a note about an object's account to an audit record.

AFP Services:

AFPAllocTemporaryDirHandle():

Maps a NetWare directory handle to an AFP directory.

AFPCreateDir():

Creates a directory with an AFP directory name.

AFPCreateFile():

Creates a file with an AFP file name.

AFPDelete():

Deletes the specified file or directory.

AFPGetBaseID():

Returns the AFP entry ID for the specified AFP file or directory.

AFPGetFileIDFromHandle():

Returns the AFP entry ID for the file specified by the NetWare handle.

AFPGetEntryIDFromPathName():

Returns an AFP entry ID for the given path name.

AFPGetFileInformation():

Returns information about the AFP side of the specified file or directory.

AFPOpenFileFork():

Opens an AFP file fork (data fork or resource fork) from a DOS environment.

AFPRename():

Moves and/or renames a file or directory.

AFPScanFileInformation();

Returns information about an AFP directory or file.

AFPSetFileInformation();

Sets information pertaining to the specified AFP file or directory.

Bindery Services:

AddBinderyObjectToSet();

Adds a bindery object to a set property.

ChangeBinderyObjectPassword();

Changes the password of a bindery object.

ChangeBinderyObjectSecurity();

Allows the supervisor to change the security of a bindery object.

ChangePropertySecurity();

Changes the security of a bindery object's property.

CloseBindery();

Allows the supervisor to close the bindery.

CreateBinderyObject();

Allows the supervisor to create a bindery object.

CreateProperty();

Adds a property to a bindery object.

DeleteBinderyObject();

Allows the supervisor to delete a bindery object.

DeleteBinderyObjectFromSet();

Deletes a bindery object from a set property.

DeleteProperty();

Deletes properties from a bindery object.

GetBinderyAccessLevel();

Returns the requesting workstation's access level to a file server's bindery.

GetBinderyObjectID();

Returns a bindery object's unique identification number.

GetBinderyObjectName();

Returns the name and type of a bindery object.

IsBinderyObjectInSet();

Determines if a bindery object is a member of a set property.

OpenBindery();

Allows the supervisor to open the bindery.

ReadPropertyValue();

Returns the value of a bindery object's item or set property.

RenameBinderyObject();

Allows the supervisor to rename a bindery object.

ScanBinderyObject();

Scans the bindery for an object.

ScanObjectTrusteePath();

Returns the directory paths to which an object has trustee rights.

ScanProperty();

Scans the bindery for an object's properties.

VerifyBinderyObjectPassword();

Verifies the password of a bindery object.

WritePropertyValue();

Writes a value to an item or set property.

Connection Services:

AttachToFileServer();

Creates an attachment between a workstation and a specified file server.

DetachFromFileServer();

Logs out the bindery object out and detaches the requesting workstation from the specified file server.

GetConnectionInformation();

Returns information about the object logged in to a specified connection.

GetConnectionNumber();

Returns the connection number that the requesting workstation uses to communicate with the default file server.

GetInternetAddress();

Returns a connection's internetwork address.

GetObjectConnectionList();

Returns a list of connection numbers that indicate how many times and under what connection numbers a bindery object is logged in to the default file server.

GetStationAddress();

Returns the physical node address of the requesting workstation.

LoginToFileServer();

Logs a bindery object in to the default file server.

Logout();

Issues a network logout request.

LogoutFromFileServer();

Logs out the object but does not detach the workstation from file server.

Workstation Services:

EndOfJob();

The shell issues this function when an application exits to automatically reset the workstation environment.

GetConnectionID();

Returns the connection ID of a file server.

GetDefaultConnectionID();

Returns the connection ID of the file server to which request packets are currently being sent.

GetFileServerName();

Returns the name of a file server.

GetNetwareShellVersion();

Returns the NetWare shell major and minor version numbers and revision level.

GetPreferredConnectionID();

Returns the connection ID of the preferred file server.

GetPrimaryConnectionID();

Returns the connection ID of the primary file server.

GetWorkstationEnvironment();

Returns information about a workstation's operating system and hardware environment.

IsConnectionIDInUse();

Determines whether a server is attached at the specified server number.

SetPreferredConnectionID();

Sets the preferred file server.

SetPrimaryConnectionID();

Sets the primary file server.

Directory Services:

AddTrusteeToDirectory();

Adds a trustee to a directory's trustee list.

AllocPermanentDirectoryHandle();

Permanently assigns a workstation drive letter to a network directory.

AllocTemporaryDirectoryHandle();

Temporarily maps a workstation drive letter to a network directory.

CreateDirectory();

Creates a directory on the file server.

DeallocateDirectoryHandle();

Deallocates a permanent or temporary directory handle.

DeleteDirectory();

Deletes a directory on the file server.

DeleteTrusteeFromDirectory();

Removes a trustee from a directory's trustee list.

GetDirectoryPath();

Returns the directory path of a directory handle.

GetEffectiveDirectoryRights();

Returns the requesting workstation's effective rights to a directory.

GetVolumeInfoWithHandle();

Given a directory handle, returns information about a volume.

GetVolumeInfoWithNumber();

Given a volume number, returns information about a volume.

GetVolumeName();

Returns a volume name for a volume.

GetVolumeNumber();

Returns the volume number for a volume.

MapDriveToPath();

Maps a path to a device name.

ModifyMaximumRightsMask();

Modifies the maximum rights mask of a directory.

RenameDirectory();

Renames a directory on the file server.

ScanDirectoryForTrustees();

Returns a directory's trustee.

ScanDirectoryInformation();

Returns information about the subdirectories below a directory.

SetDirectoryInformation();

Changes a directory's information.

StripFileServerFromPath();

Removes a file server from the front of a path.

File Services:

EraseFiles();

Erases files from a directory.

FileServerFileCopy();

Copies a file, or portion of a file, to another file on the same file server.

GetExtendedFileAttributes();

Returns a file's extended file attributes.

PurgeErasedFiles();

Permanently deletes all files that are marked for deletion.

RestoreErasedFile();

Restores one file on the file server that has been marked for deletion by the requesting workstation.

ScanFileInformation();

Returns information about a file.

SetExtendedFileAttributes();

Sets a file's extended attributes.

SetFileAttributes();

Sets a file's attributes.

SetFileInformation();

Sets file information for a file on the server.

Queue Services:

AbortServicingQueueJobAndFile();

Used to abort the servicing of a job, closes the associated file, and removes the job entry from the queue.

AttachQueueServerToQueue();

Attaches a station to a queue as a queue (job) server.

ChangeQueueJobEntry();

Changes information in a job's record entry.

ChangeQueueJobPosition();

Changes a job's position in a queue.

ChangeToClientRights();

Allows a queue (job) server to assume the login identity of the client that placed the job in the queue.

CloseFileAndAbortQueueJob();

Removes a job from a queue and closes the associated file.

CloseFileAndStartQueueJob();

Closes an associated file and releases the job for servicing.

CreateQueue();

Creates a new queue on a file server.

CreateQueueJobAndFile();

Places a new job in a queue.

DestroyQueue();

Removes a queue from the bindery and file system of a file server.

DetachQueueServerFromQueue();

Removes the requesting station from the queue's list of active queue (job) servers.

FinishServicingQueueJobAndFile();

Allows a queue (job) server to signal QMS when it has completed a job.

GetQueueJobsFileSize();

Finds the size of the associated file for a job queue.

GetQueueJobList();

Provides a list of all jobs contained in a queue.

ReadQueueJobEntry();

Retrieves information about a job in a queue.

ReadQueueCurrentStatus();

Reads the current status of a queue.

ReadQueueServerCurrentStatus();

Reads the current status record of an attached queue (job) server.

RemoveJobFromQueue();

Removes a job from a queue.

SetQueueCurrentStatus();

Controls the addition of jobs and job servers to a queue by setting or clearing bits in the queueStatus byte.

NOTE: Some Novell IBM PC QMS calls are not listed. Those unlisted calls are currently unsupported.

Synchronization Services:

ClearFile();

Unlocks the specified file and removes it from the log table of the requesting workstation.

ClearFileSet();

Unlocks and removes all files in the log table of the requesting workstation.

CloseSemaphore();

Closes a semaphore.

ExamineSemaphore();

Returns the current value and open count for a semaphore.

LockPhysicalRecordSet();

Attempts to lock all physical records in the log table of the requesting workstation.

LogPhysicalRecord();

Logs a physical record into the log table of the requesting workstation and, optionally, locks the record.

OpenSemaphore();

Opens the specified semaphore or creates it if it doesn't exist.

ReleasePhysicalRecord();

Unlocks a physical record currently locked in the log table of the requesting workstation, but does not remove it from the log table.

ReleasePhysicalRecordSet();

Unlocks all physical records currently locked in the log table of the requesting workstation, but does not remove them from the log table.

SignalSemaphore();

Increments the value of a semaphore.

WaitOnSemaphore();

Decrements the value of a semaphore.

NOTE: Some Novell IBM PC Synchronization calls are not listed. Those unlisted calls are currently unsupported.

Transaction Tracking Services:

TTSAbortTransaction();

Aborts explicit and implicit transactions.

TTSBeginTransaction();

Begins an explicit transaction.

TTSEndTransaction();

Ends an explicit or implicit transaction and returns a transaction reference number.

TTSGetApplicationThresholds();

Returns application thresholds for implicit transactions.

TTSGetWorkstationThresholds();

Returns workstation thresholds for implicit transactions.

TTSIsAvailable();

Verifies whether the default file server supports transaction tracking.

TTSSetApplicationThresholds();

Allows an application to set the number of record locks it can perform without starting an implicit transaction.

TTSSetWorkstationThresholds();

Sets workstation thresholds for implicit transactions.

GLOSSARY

(C) 1990 Oxxi, Inc.

GLOSSARY

Terms included in this glossary are defined as they are used in the Novell NetWare and AmigaDOS Manuals. Consult a standard computer dictionary for definitions of other terms.

- APF** AppleTalk Filing Protocol; a file format which allows the file server to handle 32-character names containing the full range of Apple- (and Amiga-) allowable characters. The AppleTalk Filing Protocol uses two directory entries for each file handled, one for the resource fork and one for the data fork.
- attach** To access a file server, particularly to access additional file servers after having already logged in to one file server.
- bindery** A special-purpose database maintained by the file server's operating system, used to monitor the resources and clients available on the Novell NetWare network. The bindery contains a list of "objects" (users, groups, and file servers) and their "properties" (rights, passwords, network addresses, etc.).
- bridge** A software and hardware connection between two networks, usually of similar design. A NetWare bridge can connect networks that use different kinds of network boards or transmission media, as long as both sides of the connection use the IPX protocol. If a bridge is located in a file server, it is an internal bridge; if located in a workstation, it is an external bridge.
- cache** To read data into a cache buffer in memory so that the data is available the next time it is needed and does not have to be read from the disk again. Caching greatly increases file server speed, since data in memory can be accessed up to 100 times faster than data on disk.
- client** Any personal computer connected to a Novell NetWare file server. *Also:* workstation.
- default server** The file server to which your default drive is mapped. In other words, the drive you are currently using is mapped or assigned to a particular file server, and that file server is your default server. Any Novell NetWare commands you enter will be directly automatically to the default file server *unless you specify otherwise.*

directory entries	In a NetWare volume, information stored in the volume's directory table, usually a directory name or filename. A directory's trustee list can also take up one or more directory entries, depending on how large the list is. The maximum number of directory entries that can be created on a volume is specified during Novell NetWare installation. Amiga files require an additional directory entry on the file server because they are allocated a resource fork by the AppleTalk Filing Protocol.
effective rights	The rights a user may exercise in a directory. Two factors determine effective rights: the trustee rights given a particular user, and the directory rights specified in the directory's maximum rights mask. Directory rights take precedence over trustee rights.
file attributes	Distinct from the file attributes of a file under AmigaDOS, the Novell NetWare file attributes regulate how a file can be handled on the network. For example, a file with the <i>Shareable</i> attribute can be accessed by more than one user at the same time. A <i>Read-Only</i> file may be read, but not altered.
file server	A computer that controls all network activity. The Novell NetWare operating system is loaded into the file server, and all shareable devices are attached to it. The file server controls access to shared devices and the system security; it also monitors station-to-server communications. A <i>dedicated</i> file server can be used only as a file server while it is on the network. A <i>non-dedicated</i> file server can be used simultaneously as a file server and a workstation.
gateway	A hardware/software package that allows communication between dissimilar protocols (for example, NetWare and non-NetWare networks) using industry standard protocols.
group access	A method of granting equal rights to several users at the same time so that they can all access the same directories. Rather than tediously assigning the same rights to each of a number of individual users, the network supervisor can make each user a member of the same group, then assign that group the needed rights.
home directory	A network directory that the network supervisor creates specifically for a user. The supervisor may include a drive-mapping or assignment statement to this home directory in the user's login script.

internetwork	Two or more networks connected by an internal or external bridge. Users on an internetwork can use the resources (such as data and applications files, printers, or disk drives) of all connected networks.
IPX	Internet Packet Exchange. A protocol that allows the exchange of message packets on an internetwork. With IPX, applications running on a Novell NetWare client (workstation) can use the NetWare network drivers to communicate directly with other workstations, servers or devices on the internetwork. IPX is based on Xerox Corporation's Internetwork Packet Protocol.
LAN	Local Area Network. Novell NetWare networks are one type of LAN.
log in	(verb) To gain access to the network. Logging in to the network involves executing a login script and establishing yourself as a user.
login	(noun) The process of accessing the network
login script	The set of instructions that directs your workstation to perform specific actions when you log in to the network. A system-wide login script instructs all workstations to perform the same actions upon login; an individual login script executes after the system login script, and instructs only the individual workstation. Login scripts are executed only upon <i>login</i> , not when the <i>attach</i> command is executed.
map	To assign a drive letter (or name, for the Amiga) to a chosen directory path on a particular volume of a particular file server. Essentially equal to the AmigaDOS command <i>Assign</i> .
network	A group of computers that can communicate with each other, share peripherals (such as hard disks and printers), and access remote hosts or other networks. A NetWare network consists of one or more file servers, workstations, and peripherals. NetWare network users can share the same files (both data and program files), send messages directly between individual workstations, and protect files with an extensive security system.
password protection	A security feature that requires a user to enter a correct password before being allowed to log in to the network.
peripheral	A physical device (such as a printer or disk subsystem) that is externally attached to a workstation or the network.

print server	A process that takes print jobs from the print queue and sends them to the printer. Print Servers are currently embedded in the file server.
queue	A data-handling structure that stores, in the order they are received, requests (such as print jobs) while they await servicing.
read-only	A Novell NetWare file attribute, which provides data protection by allowing the user to read, but not alter, the file.
record-locking	A feature of the Novell NetWare operating system that prevents different users from gaining simultaneous access to the same record in a shared file, thus preventing overlapping disk writes and ensuring data integrity.
resource	In NetWare installation programs, any device, feature, circuit board, or built-in circuitry that uses one or more of the following to communicate with the file server's microprocessor: interrupt lines, DMA lines, I/O addresses, or RAM or ROM memory addresses.
resource fork	The fork of a AFP file (usually Macintosh) which contains resources associated with the data. Resources can be modified only with the resource editor. Because Amiga files are handled on the file server using AFP format, each Amiga file will have a (currently empty) resource fork.
rights	Privileges (assigned by the network supervisor) that control how users can work with files in a given directory. For example, rights control whether a user may read, change or delete a file. <i>Trustee rights</i> in a directory are assigned to individual users and control what each user can do with the files in a directory and its subdirectories. <i>Directory rights</i> are assigned in the <i>maximum rights mask</i> of each individual directory and restrict the rights of all users in the directory (except the network supervisor), overriding the individual trustee rights of a user. Directory rights are limited to a single directory and do not extend down through the directory structure.
security	The control over users as they access and work with directories and files on the Novell NetWare network. There are four levels of NetWare security: login/password security, trustee security, directory security, and file attributes security.

security equivalence	A feature of network security that allows the supervisor to quickly and easily assign one user or group the same trustee rights as another user or group.
supervisor	<ol style="list-style-type: none"> 1. The network supervisor is the person responsible for the smooth operation of the whole network. (The supervisor may also install the network.) The network supervisor maintains the network, reconfiguring and updating it as the need arises. 2. SUPERVISOR is a special username automatically created when the file server is initialized. This user is permanent and cannot be deleted or renamed. The user SUPERVISOR has all rights in all file server volumes and directories, and these rights cannot be revoked. Other users or groups can be granted a security equivalence to SUPERVISOR.
TTS	Transaction Tracking System. A system that protects databases from being corrupted if the computer fails in the middle of a transaction. Each database change is regarded as one transaction, which must be either completed successfully or aborted entirely. If the workstation fails in the middle of a transaction, the transaction is "backed out" and the database is restored to its last completed state.
user	Any person who logs in to a file server.
utility	A computer program that conveniently performs one or more basic operating system tasks, such as copying files.
virtual console	A network station running the Novell NetWare FConsole utility, which allows the station to perform as a file server console, monitoring and controlling some aspect of file server activity.
workstation	Any individual personal computer connected to a Novell NetWare network and used to perform tasks through applications or utilities.

(

(

(



AmigaVision: Authoring Hints

by Cathy Godfrey

The following tips will help make your AmigaVision code more efficient. This will result in shorter loading time, faster execution of your file, and save on the amount of memory your application requires.

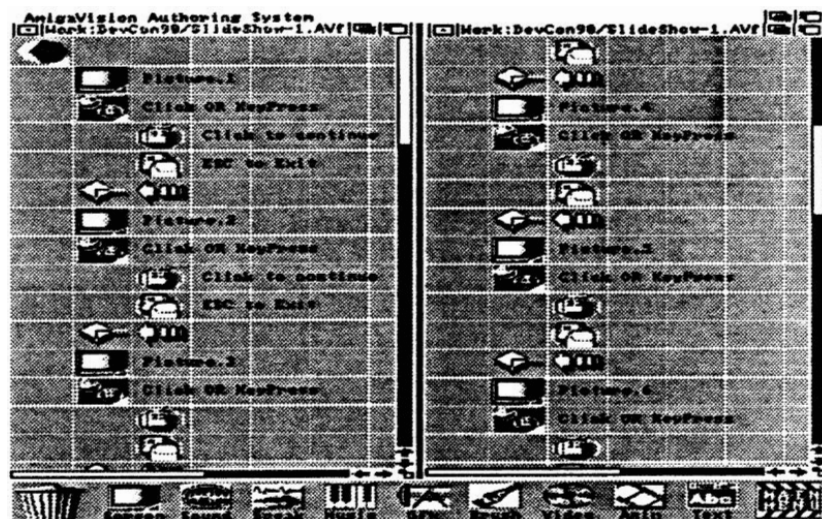
Keyboard and Mouse Interrupts

Suppose you create a very simple AmigaVision application: a slide presentation in which the user is shown a series of 35 picture files. Between each picture, the user clicks on the mouse to see the next picture or presses the Escape key to quit (see figure SlideShow-1).

This is the perfect example of where a keyboard interrupt is useful. You can replace all your Grouped Wait, Keyboard Wait, IfThen, and Quit icons with a keyboard interrupt routine.

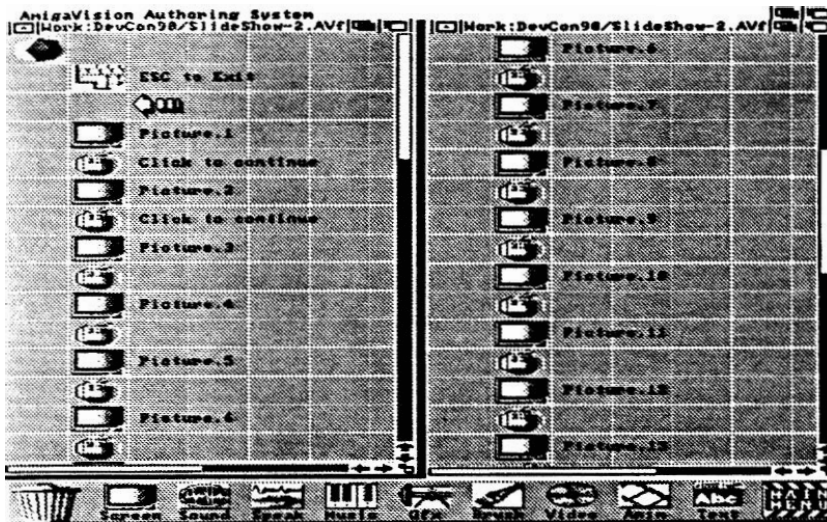
To tell AmigaVision what will activate the interrupt, you enter the name of the key(s) in the "Keys" field of the interrupt requester. For the Escape key, enter "ESC."

Once activated (when the user presses the Escape key), the keyboard interrupt will execute the actions of its children icons. For this example, we want to quit the presentation, so place a Quit icon as the child of the Keyboard Interrupt icon. Now if the Escape key is pressed, the user will quit the AmigaVision application. The resulting code looks like figure SlideShow-2.



SlideShow-1

If you require the user to click on a certain area of the screen to quit (an Exit button, perhaps), you can use a mouse interrupt in place of the keyboard interrupt.



SlideShow-2



SlideShow-3

Variables as Filenames

There is another way to further decrease the number of icons in the same slide presentation example (figure SlideShow-2). We can use a variable in the filename field of one Screen icon instead of the thirty-five separate icons we are using now.

First you must make sure that all of your picture files have the same pathname and same filename, except we will give them all a different extension. The first picture we will store in *Work:Pictures/pic.1*; the second will be in *Work:Pictures/pic.2*; then *Work:Pictures/pic.3*, etc.

Next create a variable, *count*, using the Variable icon. Initialize it to *COUNT = 0*.

Then insert a Loop icon after the Variable icon. Set it to loop for the number of pictures that you have (in this example, thirty-five).

Place a second Variable icon to increment *count* each time through the loop.

Now re-specify your Screen icon file name. Enclose the variable part of the filename in square brackets. For example, we would set the filename field to *Work:Pictures/pic.[count]*. Be sure to set the picture resolution to "File Defined."

Delete all other icons in your flow. The resulting AmigaVision flow would look like figure

SlideShow-3. The slide presentation (which was originally over 200 icons long) has now been condensed into seven icons.

One additional hint . . . in the Loop icon there is a button called "VAR." If you place a variable name in this field, this variable will always hold the current counter value at each iteration of the loop. If you use the VAR field, you don't need the second Variable icon to increment *count*. The first time through the loop, *count* will equal 1. *Work:Picture/Pic.1* will be shown. The second time, *count* equals 2, and *Work:Picture/Pic.2* will be displayed. See figure SlideShow-4 for this example code.

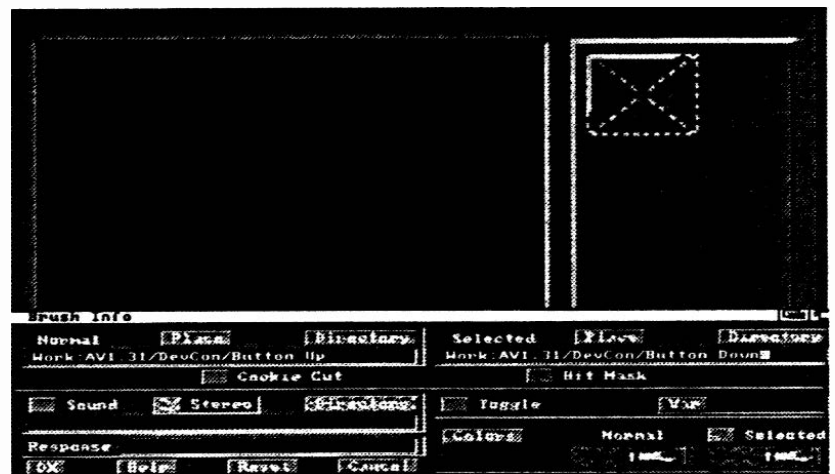


SlideShow-4

Brushes as Hit Boxes

In most cases, when you create a hit box or button on the screen, you will want it to change (in color, shape, or size) when the user clicks on it. One way to do this is to create two full-screen pictures, one with the original button and one with the selected state of the button. The disadvantage of this method is that you waste time loading in a full screen after the user activates the hit box. It would be easier to load in only the button when the hit box is clicked. AmigaVision will allow you to do this if your buttons are brushes.

In the Object Editor, instead of using a rectangle or text object as your hit box, use a brush object.



Brush-1

In the Brush Info requester there are two string gadgets. The first one holds the pathname of

the brush which is the normal image of your button, the second is for the brush in its selected state (see figure Brush-1). You can either choose two separate brushes (one for each state), or use the same brush and select different colors for each state.

Using a brush hit box will speed up your application. As soon as the hit box is activated, the user immediately sees the new brush. He/she doesn't have to wait for AmigaVision to load and display a full screen picture to get visual feedback.

General Use

These examples will explain some of the general features of AmigaVision:

1. DATABASE EXAMPLE

Figure Dbase-1 shows an example of a simple AmigaVision application which uses the built-in database feature of the authoring system. This example will allow the user to choose an artist and then view paintings created by that artist. The list of each painting and its artist is kept in the database, *Artist.dbf*.



Dbase-1

First, a brief description of the database we will use for this example: *Artist.dbf* is a database created in AmigaVision (see figure Dbase-2). Each record has two fields: ARTIST (which contains the name of the artist) and PICTURE (which contains the filename of a picture created by that artist). Listed in this database are three paintings by Renoir; three by Monet; and one by Degas.

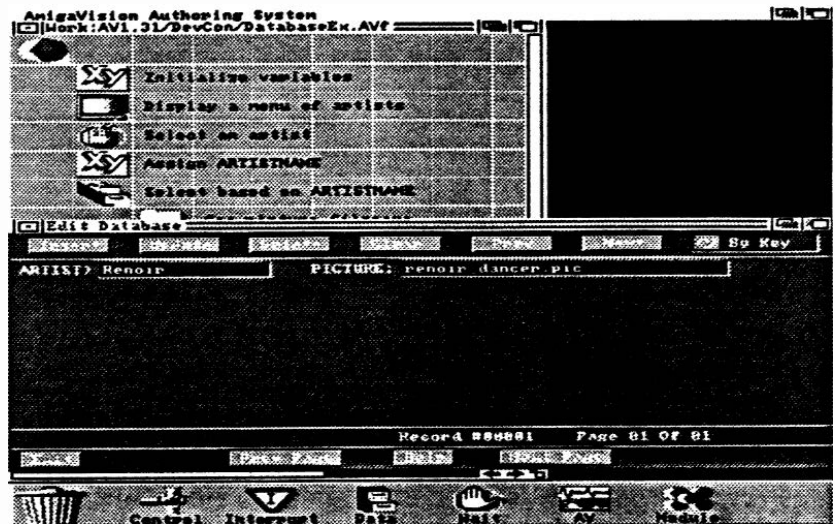
Now back to the AmigaVision flow window . . . the first Variable icon initializes two variables: ARTISTNAME and FILENAME. ARTISTNAME will keep track of which artist the user selected and FILENAME will contain the name of the picture file to be displayed.

The next two icons (Screen and Wait Mouse) set up a menu. The Screen icon displays the menu and the Wait Mouse icon sets up the hit boxes. This menu is a list of the artists that the user has to choose from. When the user clicks on an artist, the Variable icon will assign the

user's response to the variable ARTISTNAME.

The next icon is called the Select icon. It allows you to access specific records within a database. You choose these records by matching one (or more) of the fields with a variable(s). In this example we will match the field ARTIST with our variable, ARTISTNAME. Every time AmigaVision finds a record in which the field ARTIST matches the name in the variable ARTISTNAME, it will execute the actions of the children of the Select icon.

The first thing AmigaVision will do when it selects a record is to execute the Read/Write icon. This icon handles input to and output from the database. Our Read/Write icon will read the value in the PICTURE field and place it in the variable FILENAME.



Dbase-2

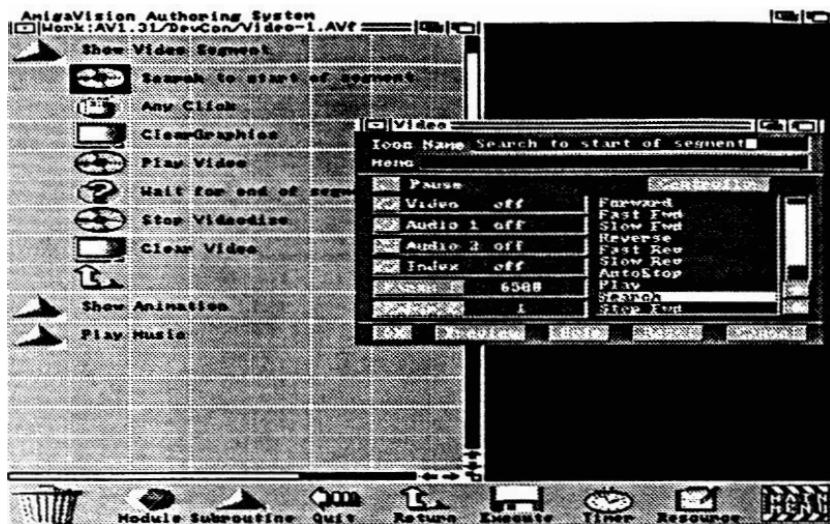
The Screen icon will use the variable FILENAME to display the picture created by the chosen artist. Then the application waits until the user clicks on the mouse.

When AmigaVision has searched the entire database for matches, it then executes the actions of the Select icon's siblings. Our database example will use a Speech icon to tell the user that they have reached the last picture. A final mouse click will end the application.

The database has many practical applications. This AmigaVision example, for instance, could easily be extended to include images stored on a videodisc. If our database had an extra field specifying frame number, we could use the Video icon to show an image of the painting stored on a videodisc. Any other information within the database (name of painting, year created, etc.) could easily be read from the database and (using the GFX icon) graphically overlayed on the video image of the painting.

2. VIDEO: USE OF THE VIDEO CONTROLLER

The Video icon allows you to program the actions of a videodisc player within your application (see figure Video-1). If you want to interactively control the player, use the



Video-1

the motion of the video. Most of these gadgets have either a right or left arrow. The right arrow plays the video forward, the left plays it backwards. For example, if you click on the right arrow gadget next to "Step," the video will advance one frame.

Beneath these controls are four gadgets that allow you to control audio, video and indexing. You can set any of these parameters ON or OFF. Note that you have independent control of both audio tracks on your videodisc. The M1 and M2 gadgets are memory areas. AmigaVision will allow you to store two separate frame numbers temporarily in memory. Click on either M1 or M2 to enter the number of the current frame of video into that memory location.

How do you get the frame numbers you've specified in the Controller back into the Video icon? First access the Controller through the Video icon requester (there is a CONTROLLER gadget for just this purpose). Then fill in the numbers of the frames you want to play (in the Start and Stop fields) or the number of the frame to which you want to search (the Frame field). Before you click on the Save button, press either the Play or Search buttons. You can save the settings for either Play or Search, but not both at the same time. You must tell AmigaVision which you wish to save by clicking on that button before you select "Save." When this is done, select "Save" and then "OK." You will then be returned to the Video requester with the proper frame numbers and videodisc action already in the requester.

Videodisc Controller (see figure VideoController-1). The Controller allows you to see the video playing as you control the player. Not only is it a nice complement to your Videodisc Log of frame numbers, it's a great tool for creating a log of your videodisc.

If you just wish to browse your videodisc, you can access the Controller from the "Tools" pull-down menu. The buttons on the top left-hand side of the requester allow you to control



VideoController-1

3. VIDEO: USE TWO BLANK SCREENS

When creating an application using a videodisc player, it is helpful to create two IFF picture files. The first file (we'll call *ClearGraphics.pic*) is used to clear all graphics from the screen and prepare the program to display video. Using your favorite paint package, create an overscan picture of the background color (or color zero). Whenever you want to clear the monitor for full-screen video, use the Screen icon to display *ClearGraphics.pic*.

There is a second useful file you can create which we'll call *ClearVideo.pic*. This is an IFF file of any color other than the background color (or color zero). When this file is displayed using the Screen icon, the video image will no longer be visible.



1

2

3



Features Outline for V2.0

by Andy Finkel

This is a brief outline of some of the new features and enhancements in Version 2.0 of the Amiga operating system. The features will be covered in more detail in the individual sessions.

ARexx

This is now a standard part of the system software. Various programs that come with the system can now use and rely on the presence of ARexx.

ASL library

This provides standard system requesters:

- ☐ a file requester
- ☐ a font requester

Battclock

Controls the Real Time Clock chip

Battmem

Controls function bits in memory of the A3000 Real Time Clock

Boot Menu

- ☐ Allows the choice of boot device/partition
- ☐ Startup-sequence disable feature

Commodities Exchange

This provides coordination and control of input handlers.

Standard commodities include:

- ☐ a window cycle commodity
- ☐ an autopoint commodity

Console

- ☐ Simple refresh character map console
- ☐ Cut andpaste from or to console

Con-handler

- ☐ Rewritten in C
- ☐ Improved editing features
- ☐ Public screen support
- ☐ Cut and paste between CON: windows
- ☐ New options: AUTO/WAIT/WINDOW/SCREEN
- ☐ Case insensitive history search
- ☐ Enhanced command line editing
- ☐ "Quiet" opening

C: commands

- ☐ Written in C
- ☐ Use the new features of the dos.library
- ☐ Smaller, faster
- ☐ Commands use pattern matching where it makes sense
- ☐ Commands use the ALL option where it makes sense
- ☐ Many commands support multiple arguments

Diskfont

- ☐ Font bitmap scaling

DOS

- ☐ Recode of the DOS library in C
- ☐ All important BCPL routines will get C interfaces
- ☐ Many new DOS functions (pattern patching, readargs, etc.)
- ☐ Really, *many* new DOS functions

Enhanced Chip Set (ECS)

ECS consists of Agnus 8372-R3 and Denise 8373-R2a. These are plug compatible replacements for the 'Far' Agnus and Denise, and may be installed in an A500 or A2000.

- ☐ *Memory Limits* – Agnus 8372-R3 allows up to 1 megabyte of CHIP memory, allowing more memory accessible by the custom chips for animation, graphics and sound applications.
- ☐ *Blitter Range* – Agnus 8372-R3 enables rectangular blits up to 32k by 32k pixels.
- ☐ *Mode Resolutions* – Installation of 8373-R2a Denise allows display of the new SuperHires mode (35 ns pixels) with up to 1280 horizontal pixels per scanline on a standard NTSC or PAL display. All the standard Amiga display resolutions and depths are still supported.
- ☐ *Monitor Scan Rates* – The V2.0 Kickstart and ECS chips support a new high-resolution Productivity mode. With the addition of a multi-sync monitor, this mode allows 640 x 480, non-interlaced screens in up to four colors.

All programs which open and operate in the Workbench screen will automatically benefit from Productivity text and graphics. In addition new programs can open their own Productivity screens in a system standard fashion.

- ☐ **Genlock Capabilities** – Denise 8373-R2a Denise allows four new genlock features:
 - ChromaKey allows any color register to control the video overlay
 - BitPlaneKey allows any bitplane to enable the video overlay
 - BorderBlank creates a transparent “frame” surrounding the active area
 - BorderNotTransparent makes an opaque “frame” surrounding the active area

Environment (ENV:)

This now uses a handler with notification that will combine small environment variables into small blocks. (also known as the 2.0 ram handler)

Exec

- ☐ Cache control functions
- ☐ Optimized autoconfig strategy
- ☐ Improved powerup strategy to manage all available memory, coprocessors and system options
- ☐ Supervisor Stack and ExecBase can be in fast memory
- ☐ More robust method of replacing system modules
- ☐ Memory and ROMs are now fully tested before use
- ☐ Memory pool manager
- ☐ Exec tests for the 68030 processor and 68882 co-processor, and enables instruction burst. Support for using the data cache and data burst has been included.
- ☐ Preparations are under way for Virtual Memory support by Exec (post 2.0).
- ☐ Performance improvements in memory allocation, signal semaphores, interrupt dispatching, and general tuning.

Expansion

- ☐ Supports Zorro III standard cards

FileSystem

- ☐ FastFileSystem in ROM that supports both DOS0 and DOS1 disks
- ☐ Notification
- ☐ Record Locking
- ☐ Links
- ☐ Variable block size
- ☐ Better disk block allocation

Graphics

- ☐ Support of new resolution modes
- ☐ Support of new monitor scan rates
- ☐ Built-in A2024 support
- ☐ BitMapScale for scaling bitmap structures (including fonts)
 - Text speed enhancements
 - ColorFont support
 - TextExtent and TextFit
 - XAttr text attribute to specify X size

Gadget Toolkit

- ☐ Provides a simple way to get and use system standard gadgets and menus
- ☐ Provides standard gadget types
- ☐ Provides a uniform appearance for programs that use the toolkit
- ☐ Provides gadget list management functions

Iffparse.library

- ☐ Shared library for parsing iff chunks

Intuition

- ☐ New Screens Support:
 - New modes
 - Scrolling/AutoScrolling
 - Overscan support
 - Public screens
- ☐ New Look windows
- ☐ New Gadgets
 - Custom gadgets
 - New string gadgets
- ☐ Custom editing
- ☐ Font selections
- ☐ Color selections
- ☐ "Activated" color selection
- ☐ "Replace/fixedfield" modes
- ☐ New State Machine
 - No more VERIFY deadlocks
 - Can call Intuition directly (pass string of InputEvents)
 - Better tablet and programmatic mouse-move support
- ☐ Independent programmable mouse and key repeat backlog limits
- ☐ EZRequester() enhancement to AutoRequest

Keymap

- ☐ usal map in ROM (a.k.a. usa)
- ☐ Separate keymap.library
- ☐ MapANSI() to complement MapRawKey()

Layers

- ☐ Better dedicing method
 - Improved damage control
 - Application backfill for Layer operations

Math IEEE Single Precision library

Alternative to MathFFP that automatically uses the 68881 if available (and other peripheral-style math chips).

Preferences

- ☐ Extensible
- ☐ More items over which the user has control
- ☐ Broken up into smaller Preference entities
- ☐ New Look (uses gadtools and asl)
- ☐ Uses ENV: handler with notification
- ☐ Support for third-party Preferences screens

Ram-handler

- ☐ Written in C
- ☐ Supports notification
- ☐ Supports "packing" of data in small files into a single block

Ramdrive

- ☐ Supports multiple units

Ramlib

- ☐ Understands the 2.0 AmigaDOS extended assign

Shell

- ☐ Built-in commands
- ☐ A bit of ARexx support in Shell. (recognizes Arexx scripts)
- ☐ Auto-CD
- ☐ Process variables and environment variables
- ☐ Variable expansion on command line

Timer

Now has some new functions, requested by popular demand.

- ☐ Wait
 - Similar in nature to the current timer wait functions
- ☐ GetMicroTics()
 - Returns the current value of an internal 64-bit counter incremented at 715909Hz (709379Hz PAL). This is a low-cost function useful for measuring short time periods.
- ☐ ReadRTC(), SetRTC(), ResetRTC()
 - Reads/Sets/Resets the time from the Commodore RTC chip.
- ☐ WaitUntil()
 - Returns when the system time is greater than or equal to the requested system time.

Trackdisk

- ☐ Improved error handling for marginal disks/drives
- ☐ No-click option on drives that support this
- ☐ Faster
- ☐ Doesn't need as much chip ram (can use fast ram for many operations)

Workbench

- ☐ New improved look and feel
- ☐ Easier to use:
 - Menu redesign into more logical arrangement.
 - Backdrop icons, select-all, drag-select, unsnapshot, shortcut keys, Workbench startup drawer, can cancel operations like drag, drag-select, select.
- ☐ Can execute any CLI command.
- ☐ Asynchronous refresh and program load
- ☐ More intelligent cleanup routine.
- ☐ Programmer callable routines
 - application menus
 - application icons
- ☐ User customizable:
 - font, colors, patterns, backdrop pattern





Workbench V2.0 Documentation Update

by Dave Berezowski

This article is an update to the Workbench documentation incorporating new information about the V2.0 release.

V2.0 Workbench User Interface

Here's a summary of the changes to Workbench for V2.0:

MENUS	Standard Workbench menus have been expanded.
MOUSE BUTTONS	Some new operations have been added.
WINDOWS	Workbench windows have been enhanced.
ICON TEXT	Icon text can now be customized.
STARTUP DRAWER	This is a "startup-sequence" for Workbench.
ASYNCHRONICITY	There is now more multitasking to Workbench.

The New Menus

Under V2.0, the Workbench, Disk, and Special menus have been replaced with Workbench, Window, and Icon menus. The Workbench menu contains menu items that generally operate on Workbench as a whole. The Window menu contains menu items that work only on the active window or icons contained in the active window. The Icons menu contains menu items that work only on icons that have been selected. Also, some menu items now have command-key (right Amiga) capability.

The Workbench Menu

This menu provides functions which are, in general, neither window nor icon specific.

(new) Backdrop	This toggles the Workbench window between a backdrop and a non-backdrop window.
(new) Execute Command	This gives you a one line CLI (with a one line history) in which virtually any CLI-type command can be executed.
Redraw All	No change here (formerly Redraw).
Update All	This updates all Workbench windows (except the Workbench disk window). Refer to 'Update' in the 'Window Menu' section below for an explanation.
Last Error	DOS error numbers now have a textual string associated with them.
Version	No change here.
(new) Quit	<p>This requests Workbench to pack up and go away completely. All resources and memory are freed. You cannot quit Workbench if either of the following is true:</p> <ol style="list-style-type: none">1. There are Workbench launched programs still running.2. Somebody, other than Workbench, is using the Workbench.library (i.e., the library open count is greater than 1). It is highly recommended that you do not quit Workbench unless you have a way of starting it up again (leave a CLI open). The easiest way to do this is to launch a newshell just before quitting Workbench. To do this, select Execute and type 'NewShell'. You'll get a shell that is detached from Workbench.

The Window Menu

This menu performs operations on the currently active window.

(new) New Drawer	This quickly creates a new drawer in the active window. The new drawer is called Unnamed n where n is a number starting at 1. Previous to this, one had to duplicate the Empty drawer to create a new drawer (which was very slow).
(new) Open Parent	This opens and makes active the parent window of the currently active window. It is the Workbench equivalent to the Parent gadget in a CLI file requester.
Close	No change here.

- (new) Update This scans the directory of the active window and refreshes the display. New icons are displayed, any deleted icons are removed. A future goal for Workbench is support for filesystems that use notification so that this type of updating can be done automatically. However, there will always be filesystems which do not support automatic updating (like those across an Ethernet) that will need this update option. Note that with the advent of automatic icon updating via PutDiskObject() in icon.library, you should not need to use this often.
- (new) Select Contents This selects all the icons in the active window.
- Cleanup Cleanup now accounts for the width/height of the text as well as the width/height of the gadget. Columns widths are variable and are computed by the widest icon in the column. The disk icons can also be cleaned up via selecting the Workbench window as the active window (see Mouse Buttons below) and then selecting this option.
- (new) Snapshot The active window's size and position are saved to disk when 'Snapshot Window' is selected. 'Snapshot All' also saves all the icons to disk.
- (new) View By Icons can now be viewed in a textual mode; sorted by Name, Date or Size. This option is 'sticky' in that the window will open in the last mode it was viewed in. In 'View By Text' mode (i.e., View By Name/Date/Size) the font used to display the information can be selected by using the preferences editor 'font' and modifying the system default text' font.
- (new) Show All This shows all the files which do not have a .info file. Selecting 'Show All Files' causes Workbench to scan the current directory. Any file found in the directory that doesn't have a corresponding '.info' file gets either a default drawer, tool or project icon created for it. The rule used to determine if a file should be displayed as a tool or a project is as follows: if the file has the execute bit set, it is considered to be a tool; otherwise, it is considered to be a project. Selecting 'Show Only Icons' causes Workbench to remove any 'fake' icons which appeared as a result of selecting 'Show All Files'. If you want to turn a 'fake' icon into a real one, drag the fake icon into IconEdit (found in the Tools directory) and select save. Note that if you do this, the icon will now be started with Workbench (vs CLI) arguments. If the program cannot accept this it may very well crash the system!

The Icon Menu

This menu performs operations on the currently selected icon(s).

Open	Any icons created by 'Show All Files' are run with CLI type arguments. This means, for instance, that one could click on the DIR command and see the results.
Copy	This has been re-written. It's much faster and doesn't use up all memory!
Rename	This is now refresh asynchronous. The Rename window now has a drag bar as well as front-to-back gadgets.
Information	This has been re-written and is now refresh asynchronous.
Snapshot	All selected icons are saved to disk.
(new) UnSnapshot	All selected icons get their position set to NO_ICON_POSITION.
(new) Leave Out	The selected icons are added to the .backdrop file and will automatically come up in the Workbench window the next time a LoadWb command is performed. They are also automatically moved to the Workbench window if they are not already there.
(new) Put Away	The selected icons are removed from the .backdrop file and will not automatically come up in the Workbench window the next time a LoadWb command is performed. They are also automatically removed from the Workbench window and placed in their own window (if open). This function also applies to those icons which have been placed in the Workbench window but not left out.
(new) -----	a separator bar.
Delete	No change here (formerly called Discard).
Format Disk	Format Disk (formerly called Initialize Disk) now puts up a requester giving you the choice to either 'Format', 'Re-format' or 'Cancel'. 'Re-format' is essentially a 'Format QUICK'.
Empty Trash	No change here.

The Tools Menu

This is an all new menu. The user menu items (AppMenuItems) will go here. Refer to the documentation on AddAppMenuItem and RemoveAppMenuItems in the AutoDoc section of this article for more info.

(new) **ResetWB** This causes Workbench to close and then re-open all of its windows, if possible. It is not possible if there are any non-Workbench or application windows open on the Workbench screen. When Workbench re-opens, it will use the new current font, font color, background pattern, etc. This function is essentially a CloseWorkbench() followed by an OpenWorkbench().

Mouse Buttons

Left Button Pressing and holding the left mouse button enters 'drag select' mode. In this mode, a dotted box is drawn and all icons which fall inside this box are selected when the button is released. Double-clicking the left mouse button inside a window brings that window to the front if that option has been enabled in the preferences editor 'WBConfig'.

Right Button Clicking the right mouse button while dragging icons cancels the operation. Clicking the right mouse button while drag-selecting cancels the operation. Clicking the right mouse button while selecting icons cancels that selection. If you want to select all the icons in a window bar one, you can do the following:

1. Chose 'Select All'
2. While holding down a shift-key, press the left mouse button on the icon you want to exclude and then click the right mouse button. The icon is now de-selected and the rest of the icons are still selected.

Windows

Workbench Window (formerly the backdrop window)

This is now just another Workbench window. It can be selected by clicking the left mouse button. The backdrop window pattern can be selected by the user via Preferences. The Workbench pattern is set to all dots off for this release. Use the preferences editor 'wbpattern' to change it. Workbench can be toggled back to a backdrop window by selecting the 'Backdrop' menu item from the Workbench menu. You can customize the power up mode of the Workbench window in the preferences editor 'WBConfig'.

Drawer Windows

These now have cleaner arrow gadget images and the arrows have been moved to the lower right for convenience. The drawer pattern is now user selectable via Preferences. The Window pattern is set to all dots off for this release. Use the Preferences editor 'wbpattern' to change it. Drawer windows are now active when opened. The fuel gauge is gone. It has been replaced by a string in the title bar which displays disk usage (only for disk icon windows).

Icon Text

The font type and size for an icon is now user selectable via Preferences. The font color and draw mode (JAM1 or JAM2) are also selectable by using the Preferences editor 'font' and modifying the 'Workbench icon text' font.

Startup Drawer

Any icon found in the startup drawer 'wbstartup' gets run when Workbench is invoked. There are several new tooltypes which can be used to modify the way these icons are executed. They are:

CLI	This tells Workbench to run this program with CLI arguments (as opposed to Workbench arguments)
STARTPRI=n	This allows you to specify the priority in which icons will be launched. The default priority is 0, the valid range is -128 to +127.
DONOTWAIT	This tells Workbench not to wait for this icon to finish executing before launching the next icon.
WAIT=n	This allows one to specify a minimum time (in seconds) for Workbench to wait before continuing on with the next icon.

Asynchronicity

Rename and Info are now refresh asynchronous. Refresh Events are also asynchronous. For example, if Workbench is opening a drawer and gets a refresh-window event, it will now process the event immediately as opposed to waiting for the drawer opening to complete.

Workbench loading is also asynchronous; it now asynchronously loads the tool you double-click on. While Workbench is attempting to load your tool the message "Attempting to load program 'program_name'..." appears in the screen title bar.

Other Workbench Changes

Here are a few other Workbench changes:

- ☐ The screen title bar now displays how much graphic and other memory is available.
- ☐ Workbench now searches the path from the CLI it was invoked from if it cannot find an icon's tool.
- ☐ Setting a task's priority via a ToolType of TOOLPRI now works.
- ☐ Workbench no longer passes a NULL lock for project icons that specify the default tool as an absolute path, which was incorrect. (i.e., if the default tool is 'volume:dir/tool', Workbench will now pass a lock to 'volume:dir' and a filename of 'tool').
- ☐ Workbench now accepts (internal) messages from icon.library when an icon has been written to disk via the icon.library call PutDiskObject() or PutDefDiskObject(). If the window in which the icon lives is open, that icon will be created/updated.
- ☐ It is no longer possible for the system to lock up due to Workbench having the layers locked and another process locking layers as well. For example, in the V1.3 release, if you were dragging an icon, and another process did a WindowToFront(), the machine would lock up. Workbench 2.0 has a layers daemon which looks for such a situation. If it detects a lock up, it will abort the operation (which unlocks the layer) thereby unfreezing the machine. If you are ever drag selecting or dragging icons and the system appears to freeze for a moment and aborts the operation, you'll know that Workbench has just prevented the system from locking up.

V2.0 Workbench Programmer Interface

As of V2.0, Workbench is a callable library. Six calls have been implemented for the V2.0 release. These are covered in detail in the accompanying autodoc contained in this document.

V2.0 Workbench Test Suite

Included on the release disk is a suite of programs for testing the new Workbench features. These programs must be run from the CLI. These programs open a tiny window in the top left corner of the Workbench screen to let you know that they are running. Select the CloseWindow gadget in the appropriate window to remove the AppIcon, AppMenuItem or AppWindow.

AppIcon

This is a test of the Application Icon interface. It includes calls to AddAppIcon() and RemoveAppIcon(). When run, an icon named AppIcon will appear on the Workbench. Double-clicking or dropping icons on it will cause Workbench to send a message to the application which will print (in the CLI window) information about the action. The data (image) for the AppIcon is contained in the file 'appicon.image'. Alternatively, the data could have been created via calling GetDiskObject or GetDefDiskObject in icon.library.

AppMenuItem

This is a test of the Application MenuItem interface. It includes calls to AddAppMenuItem() and RemoveAppMenuItem(). When run, two menu items (named appmenuItem1 and appmenuItem2) will be added to the Workbench Tools menu. Selecting either one of these menu items will cause Workbench to send a message to the application which will print information (in the CLI window) about which menuItem was selected.

AppWindow

This is a test of the Application Window interface. It includes calls to AddAppWindow() and RemoveAppWindow(). When run, a window will open in the top left corner of the Workbench screen. If icons are dropped on this window, Workbench will send a message to the application which will print (in the CLI window) information about the icon(s).

Workbench AutoDocs

The autodocs for the new Workbench functions are listed on the following pages.

TABLE OF CONTENTS

workbench.library/AddAppIcon
workbench.library/AddAppMenuItem
workbench.library/AddAppWindow
workbench.library/RemoveAppIcon
workbench.library/RemoveAppMenuItem
workbench.library/RemoveAppWindow

NAME

AddAppIcon - add an icon to workbench's list of appicons. (V36)

SYNOPSIS

```
AppIcon = AddAppIcon(id, userdata, text, msgport, lock, diskobj, taglist)
D0          D0      D1      A0      A1      A2      A3      A4
struct AppIcon *AddAppIcon(ULONG, ULONG, char *, struct MsgPort *,
    struct FileLock *, struct DiskObject *, struct TagItem *);
```

FUNCTION

Attempts to add an icon to workbench's list of appicons. If successful, the icon is displayed on the workbench window (the same place disk icons are displayed). This call is provided to allow applications to be notified when a graphical object (not necessarily associated with a file) gets 'manipulated'. (explained later). The notification consists of an AppMessage (found in workbench.h/i) of type 'MTYPE_APPICON' arriving at the message port you specified. The types of 'manipulation' that can occur are:

1. Double-clicking on the icon. am_NumArgs will be zero and am_ArgList will be NULL.
2. Dropping an icon or icons on your appicon. am_NumArgs will be the number of icons dropped on your appicon plus one. am_ArgList will be an array of ptrs to WBArg structures. Refer to the 'WBStartup Message' section of the RKM for more info.
3. Dropping your appicon on another icon. NOT SUPPORTED.

INPUTS

id - this variable is strictly for your own use and is ignored by workbench. Typical uses in C are in switch and case statements, and in assembly language table lookup.

userdata - this variable is strictly for your own use and is ignored by workbench.

text - name of icon (char *)

lock - lock on the file you want associated with the icon, NULL for icons which have no file.

msgport - pointer to message port workbench will use to send you an AppMessage message of type 'MTYPE_APPICON' when your icon gets 'manipulated' (explained above).
- if NULL, DefaultTool is invoked. NOT IMPLEMENTED!

diskobj - pointer to a DiskObject structure filled in as follows:

- do_Magic - NULL
- do_Version - NULL
- do_Gadget - ptr to gadget structure filled in as follows:
 - NextGadget - NULL
 - LeftEdge - NULL
 - TopEdge - NULL
 - Width - width of icon hit-box
 - Height - height of icon hit-box
 - Flags - NULL
 - Activation - NULL
 - GadgetType - NULL

GadgetRender - pointer to Image structure filled in as follows:

- LeftEdge - NULL
- TopEdge - NULL
- Width - width of image
- Height - height of image
- Depth - # of bit-planes in image
- ImageData - pointer to actual word aligned bits (CHIP MEM)
- PlanePick - specific to image
- PlaneOnOff - specific to image
- NextImage - NULL
- SelectRender - pointer to alternate Image struct or NULL
- GadgetText - NULL
- MutualExclude - NULL
- SpecialInfo - NULL
- GadgetID - NULL
- UserData - NULL
- do_Type - NULL
- do_DefaultTool - pointer to default tool string for this icon
or NULL if there is no default tool.
- do_ToolTypes - array of pointers to tooltype strings for this
icon or NULL if there are no tooltype strings.
- do_CurrentX - NO_ICON_POSITION (recommended)
- do_CurrentY - NO_ICON_POSITION (recommended)
- do_DrawerData - NULL
- do_ToolWindow - pointer to toolwindow string for icon
or NULL if there is no toolwindow.
- do_StackSize - stacksize required if DefaultTool is to be invoked
(an easy way to create one of these (a DiskObject) is to create an icon
with the V2.0 icon editor and save it out. Your application can then
call GetDiskObject on it and pass that to AddAppIcon.)

taglist - ptr to a list of tag items. Must be NULL for V2.0.

RESULTS

AppIcon - a pointer to an appicon structure which you pass to
RemoveAppIcon when you want to remove the icon
from workbench's list of appicons. NULL
if workbench was unable to add your icon; typically
happens under low memory conditions.

EXAMPLE

You could design a print-spooler icon and add it to the workbench.
Any file dropped on the print spooler would be printed. If the
user double-clicked (opened) your printer-spooler icon, you could
open a window showing the status of the print spool, allow changes
to print priorities, allow deletions, etc. If you registered this
window as an 'appwindow' (explained in workbench.library/AddAppWindow)
files could also be dropped in the window and added to the spool.

SEE ALSO

RemoveAppIcon

BUGS

Currently Info cannot be obtained on appicons.

NAME

AddAppMenuItem - add a menuitem to workbench's list of appmenuitems. (V36)

SYNOPSIS

```
AppMenuItem = AddAppMenuItem(id, userdata, text, msgport, taglist)
      D0                D0      D1      A0      A1      A2
struct AppMenuItem *AddAppMenuItem(ULONG, ULONG, char *,
      struct MsgPort *, struct TagItem *);
```

FUNCTION

Attempt to add the text as a menuitem to workbench's list of appmenuitems (the 'Tools' menu strip).

INPUTS

id - this variable is strictly for your own use and is ignored by workbench. Typical uses in C are in switch and case statements, and in assembly language table lookup.

userdata - this variable is strictly for your own use and is ignored by workbench.

text - text for the menuitem (char *)

msgport - pointer to message port workbench will use to send you an AppMessage message of type 'MTYPE_APPMENUITEM' when your menuitem gets selected.

- if NULL, 'text' is invoked directly as a program. (NOT YET IMPLEMENTED! Will probably need to pass me a lock or a path so I can LoadSeg the program).

taglist - ptr to a list of tag items. Must be NULL for V2.0.

RESULTS

AppMenuItem - a pointer to an appmenuitem structure which you pass to RemoveAppMenuItem when you want to remove the menuitem from workbench's list of appmenuitems. NULL if workbench was unable to add your menuitem; typically happens under low memory conditions.

SEE ALSO

RemoveAppMenuItem

BUGS

Does not handle menuitems running off the bottom of the screen. ie. you can currently add too many menuitems and possibly crash.

NAME

AddAppWindow - add a window to workbench's list of appwindows. (V36)

SYNOPSIS

```
AppWindow = AddAppWindow(id, userdata, window, msgport, taglist)
      D0          D0    D1      A0      A1      A2
struct AppWindow *AddAppWindow(ULONG, ULONG, struct Window *,
                               struct MsgPort *, struct TagItem *);
```

FUNCTION

Attempt to add the window to workbench's list of appwindows. Normally non-workbench windows (those not opened by workbench) cannot have icons dropped in them. This call is provided to allow applications to be notified when an icon or icons get dropped inside a window that they have registered with workbench. The notification consists of an AppMessage (found in workbench.h/1) of type 'MTYPE_APPWINDOW' arriving at the message port you specified. What you do with the list of icons (pointed to by am_ArgList) is up to you, but generally you would want to call GetDiskObjectNew on them.

INPUTS

id - this variable is strictly for your own use and is ignored by workbench. Typical uses in C are in switch and case statements, and in assembly language table lookup.
 userdata - this variable is strictly for your own use and is ignored by workbench.
 window - pointer to window to add.
 msgport - pointer to message port workbench will use to send you an AppMessage message of type 'MTYPE_APPWINDOW' when your window gets an icon or icons dropped in it.
 taglist - ptr to a list of tag items. Must be NULL for V2.0.

RESULTS

AppWindow - a pointer to an appwindow structure which you pass to RemoveAppWindow when you want to remove the window from workbench's list of appwindows. NULL if workbench was unable to add your window; typically happens under low memory conditions.

SEE ALSO

RemoveAppWindow

The V2.0 icon editor is an example of an app window. Note that app window applications generally want to call GetDiskObjectNew (as opposed to GetDiskObject) to get the disk object for the icon dropped in the window.

BUGS

None

workbench.library/RemoveAppIcon

workbench.library/RemoveAppIcon

NAME

RemoveAppIcon - remove an icon from workbench's list of appicons. (V36)

SYNOPSIS

```
error = RemoveAppIcon(AppIcon)
DO      A0
BOOL RemoveAppIcon(struct AppIcon *);
```

FUNCTION

Attempt to remove an appicon from workbench's list of appicons.

INPUTS

AppIcon - pointer to an AppIcon structure returned by AddAppIcon.

RESULTS

error - TRUE if successful, else FALSE. Typically 'FALSE' would only be returned if you tried to remove an appicon that was not on workbench's list of appicons.

SEE ALSO

AddAppIcon

BUGS

None

workbench.library/RemoveAppMenuItem

workbench.library/RemoveAppMenuItem

NAME

RemoveAppMenuItem - remove a menuitem from workbench's list of appmenuitems. (V36)

SYNOPSIS

```
error = RemoveAppMenuItem(AppMenuItem)
      DO                      A0
      BOOL RemoveAppMenuItem(struct AppMenuItem *);
```

FUNCTION

Attempt to remove an appmenuitem from workbench's list of appmenuitems.

INPUTS

AppMenuItem - pointer to an AppMenuItem structure returned by AddAppMenuItem.

RESULTS

error - TRUE if successful, else FALSE. Typically 'FALSE' would only be returned if you tried to remove an appmenuitem that was not on workbench's list of appmenuitems.

SEE ALSO

AddAppMenuItem

BUGS

None

workbench.library/RemoveAppWindow

workbench.library/RemoveAppWindow

NAME

RemoveAppWindow - remove a window from workbench's list of appwindows. (V36)

SYNOPSIS

```
error = RemoveAppWindow(AppWindow)
DO                                A0
BOOL RemoveAppWindow(struct AppWindow *);
```

FUNCTION

Attempt to remove an appwindow from workbench's list of appwindows.

INPUTS

AppWindow - pointer to an AppWindow structure returned by AddAppWindow.

RESULTS

error - TRUE if successful, else FALSE. Typically 'FALSE' would only be returned if you tried to remove an appwindow that was not on workbench's list of appwindows.

SEE ALSO

AddAppWindow

BUGS

None

Icon Library V2.0

by Dave Berezowski

This article is an update to the Icon Library documentation incorporating new information about the V2.0 release. The information presented here applies to the V2.0 release and is contained in three sections; bug fixes, enhancements, and new library calls.

V2.0 Icon Library fixes include:

- ☐ icon.library now passes enforcer, memmung and io_torture. This means that it no longer accesses inappropriate memory locations or re-uses memory after it as freed it.
- ☐ the use of MEMF_CHIP type memory has been reduced.
- ☐ PutWBObject() now closes the file it was writing if it encounters an error.

V2.0 Icon Library enhancements:

- ☐ BumpRevision(), FindToolType() and MatchToolValue() are now case insensitive. This change make it easier for the user and programmer to enter or find the correct tool type and match tool values.
- ☐ FindToolType() no longer requires an equal sign in the statement. Thus a tool type of 'DONOTWAIT' is now recognized whereas under V1.3 one would have had to enter 'DONOTWAIT='.
- ☐ BumpRevision() now uses the underscore character (instead of a space) when composing names. It is, however, backwards compatable with V1.3. The use of the underscore character removes the burden of enclosing the filename in quotes as one must do when a filename contains spaces.

- ❑ PutDiskObject() now notifies Workbench (via an internal mechanism) if the operation was successful. This feature has been coined 'Automatic Icon Updating'. After a successful PutDiskObject(), Workbench will receive the message and either create a new icon or update the old one if the window in which the icon would live is open. If Workbench is not running (i.e., a LoadWb has not yet been performed) then the notification is bypassed.

Summary of new library calls for V2.0 icon.library:

GetDefDiskObject() - read default Workbench disk object from disk. Refer to the icon.library autodocs below for a complete description.

PutDefDiskObject() - write disk object as the default for its type. Refer to the icon.library autodocs below for a complete description.

GetDiskObjectNew() - read in a Workbench disk object from disk. This is the preferred call (over GetDiskObject) for those applications using the AddAppWindow() library call of workbench.library. Refer to the icon.library autodocs below for a complete description.

icon.library Autodocs

TABLE OF CONTENTS

icon.library/AddFreeList
icon.library/BumpRevision
icon.library/FindToolType
icon.library/FreeDiskObject
icon.library/FreeFreeList
icon.library/GetDefDiskObject
icon.library/GetDiskObject
icon.library/GetDiskObjectNew
icon.library/MatchToolValue
icon.library/PutDefDiskObject
icon.library/PutDiskObject

TABLE OF CONTENTS

Icon.library/AddFreeList
Icon.library/BumpRevision
Icon.library/FindToolType
Icon.library/FreeDiskObject
Icon.library/FreeFreeList
Icon.library/GetDefDiskObject
Icon.library/GetDiskObject
Icon.library/GetDiskObjectNew
Icon.library/MatchToolValue
Icon.library/PutDefDiskObject
Icon.library/PutDiskObject

Icon.library/AddFreeList

Icon.library/AddFreeList

NAME

AddFreeList - add memory to a free list.

SYNOPSIS

status = AddFreeList(free, mem, len)
DO A0 A1 A2
BOOL AddFreeList(struct FreeList *, APTR, ULONG);

FUNCTION

This routine adds the specified memory to the free list. The free list will be extended (if required). If there is not enough memory to complete the call, a null is returned.

Note that AddFreeList does NOT allocate the requested memory. It only records the memory in the free list.

INPUTS

free -- a pointer to a FreeList structure
mem -- the base of the memory to be recorded
len -- the length of the memory to be recorded

RESULTS

status -- TRUE if the call succeeded else FALSE;

SEE ALSO
AllocEntry, FreeEntry, FreeFreeList

BUGS

None

Icon.library/BumpRevision

Icon.library/BumpRevision

NAME

BumpRevision - reformat a name for a second copy.

SYNOPSIS

```
result = BumpRevision(newbuf, oldname)
DO
  A0
  A1
char *BumpRevision(char *, char *);
```

FUNCTION

BumpRevision takes a name and turns it into a "copy of name". It knows how to deal with copies of copies. The routine will truncate the new name to the maximum dos name size (currently 30 characters).

INPUTS

newbuf - the new buffer that will receive the name (it must be at least 31 characters long).
oldname - the original name

RESULTS

result - a pointer to newbuf

EXAMPLE

```
oldname      newbuf
-----
"foo"        "copy of foo"
"copy_of_foo" "copy_2_of_foo"
"copy_2_of_foo" "copy_3_of_foo"
"copy_3_of_foo" "copy_200_of_foo"
"copy_200_of_foo" "copy_of_copy_foo"
"copy_of_copy_foo" "copy_1_of_foo"
"copy_1_of_foo" "copy_of_01234567890123456789"
"copy_of_01234567890123456789" "copy_of_0123456789012345678901"
```

SEE ALSO

BUGS

None

Icon.library/FindToolType

Icon.library/FindToolType

NAME

FindToolType - find the value of a ToolType variable.

SYNOPSIS

```
value = FindToolType(toolTypeArray, typeName)
DO
  A0
  A1
char *FindToolType(char **, char *);
```

FUNCTION

This function searches a tool type array for a given entry, and returns a pointer to that entry. This is useful for finding standard tool type variables. The returned value is not a new copy of the string but is only a pointer to the part of the string after typeName.

INPUTS

toolTypeArray - an array of strings (char **).
typeName - the name of the tooltype entry (char *).

RESULTS

value - a pointer to a string that is the value bound to typeName, or NULL if typeName is not in the toolTypeArray.

EXAMPLE

```
Assume the tool type array has two strings in it:
"FILETYPE=TEXT"
"TEMPDIR=it"

FindToolType(toolTypeArray, "FILETYPE") returns "TEXT"
FindToolType(toolTypeArray, "FILETYPE") returns "TEXT"
FindToolType(toolTypeArray, "TEMPDIR") returns "it"
FindToolType(toolTypeArray, "MAXSIZE") returns NULL
```

SEE ALSO

MatchToolValue

BUGS

None

Icon.library/FreeDiskObject

Icon.library/FreeDiskObject

NAME

FreeDiskObject - free all memory in a Workbench disk object.

SYNOPSIS

FreeDiskObject(diskobj)

AO

void FreeDiskObject(struct DiskObject *);

FUNCTION

This routine frees all memory in a Workbench disk object, and the object itself. It is implemented via FreeFreeList().

GetDiskObject() takes care of all the initialization required to set up the object's free list. This procedure may ONLY be called on a DiskObject allocated via GetDiskObject().

INPUTS

diskobj -- a pointer to a DiskObject structure

RESULTS

None

SEE ALSO

GetDiskObject, FreeFreeList

BUGS

None

Icon.library/FreeFreeList

Icon.library/FreeFreeList

NAME

FreeFreeList - free all memory in a free list.

SYNOPSIS

FreeFreeList(free)

AO

void FreeFreeList(struct FreeList *);

FUNCTION

This routine frees all memory in a free list, and the free list itself. It is useful for easily getting rid of all memory in a series of structures. There is a free list in a Workbench object, and this contains all the memory associated with that object.

A FreeList is a list of MemList structures. See the MemList and MemEntry documentation for more information.

If the FreeList itself is in the free list, it must be in the first MemList in the FreeList.

INPUTS

free -- a pointer to a FreeList structure

RESULTS

None

SEE ALSO

AllocEntry, FreeEntry, AddFreeList

BUGS

None

Icon.library/GetDefDiskObject

Icon.library/GetDefDiskObject

NAME

GetDefDiskObject - read default wb disk object from disk.

(V36)

SYNOPSIS

diskobj = GetDefDiskObject(def_type)

DO

struct DiskObject *GetDiskObject(LONG);

FUNCTION

This routine reads in a default Workbench disk object in from disk. The valid def types can be found in workbench/workbench.h and currently include WBDISK thru WBOARBOX. If the call fails, it will return zero. The reason for the failure may be obtained via IoErr().

Using this routine protects you from any future changes to the way default Icons are stored within the system.

INPUTS

def_type - default Icon type (WBDISK thru WBKICK). Note that the define 'WBDEVICE' is not currently supported.

RESULTS

diskobj -- the default Workbench disk object in question

SEE ALSO

PutDefDiskObject

BUGS

None

Icon.library/GetDiskObject

Icon.library/GetDiskObject

NAME

GetDiskObject - read in a Workbench disk object from disk.

SYNOPSIS

diskobj = GetDiskObject(name)

DO

struct DiskObject *GetDiskObject(char *);

FUNCTION

This routine reads in a Workbench disk object in from disk. The name parameter will have a "-info" postpended to it, and the info file of that name will be read. If the call fails, it will return zero. The reason for the failure may be obtained via IoErr().

Using this routine protects you from any future changes to the way Icons are stored within the system.

A freelist structure is allocated just after the DiskObject structure. FreeDiskObject makes use of this to get rid of the memory that was allocated.

INPUTS

name -- name of the object (char *) or NULL. If you just want a DiskObject structure allocated for you (useful when calling AddAppIcon in workbench.library).

RESULTS

diskobj -- the Workbench disk object in question

SEE ALSO

FreeDiskObject

BUGS

None

```

Icon.library/GetDiskObjectNew      Icon.library/GetDiskObjectNew

NAME
  GetDiskObjectNew - read in a Workbench disk object from disk.      (V36)

SYNOPSIS
  diskobj = GetDiskObjectNew(name)
  DO
  struct DiskObject *GetDiskObject(char *);

FUNCTION
  This routine reads in a Workbench disk object in from disk. The
  name parameter will have a ".info" postpended to it, and the
  info file of that name will be read. If the call fails,
  it will return zero. The reason for the failure may be obtained
  via IOErr().

  Using this routine protects you from any future changes to
  the way Icons are stored within the system.

  A FreeList structure is allocated just after the DiskObject
  structure; FreeDiskObject makes use of this to get rid of the
  memory that was allocated.

  This call is functionally identical to GetDiskObject with one exception.
  If its call to GetDiskObject fails, this function calls GetDefDiskObject.
  This is useful when there is no .info file for the icon you are trying
  to get a disk object for. Applications that use Workbench application
  windows MUST use this call if they want to handle the user dropping an
  icon (that doesn't have a .info file) on their window. The V2.0
  icon editor program is an example of a Workbench application window
  that uses this call.

INPUTS
  name -- name of the object (char *) or NULL. If you just want a
  DiskObject structure allocated for you (useful when
  calling AddAppIcon in Workbench.library).

RESULTS
  diskobj -- the Workbench disk object in question

SEE ALSO
  FreeDiskObject

BUGS
  None

```

```

Icon.library/MatchToolValue      Icon.library/MatchToolValue

NAME
  MatchToolValue - check a tool type variable for a particular value.

SYNOPSIS
  result = MatchToolValue(toolType, value)
  DO
  BOOL MatchToolValue(char *, char *);

FUNCTION
  MatchToolValue is useful for parsing a tool type value for
  a known value. It knows how to parse the syntax for a tool
  type value (in particular, it knows that '|' separates
  alternate values). Note that the parsing is case insensitive.

INPUTS
  toolType - a ToolType value (as returned by FindToolType)
  value - you are interested if value appears in toolType

RESULTS
  result - TRUE if the value was in toolType else FALSE.

EXAMPLE
  Assume there are two type strings:
  type1 = "text"
  type2 = "albic"

  MatchToolValue( type1, "text" ) returns TRUE
  MatchToolValue( type1, "data" ) returns FALSE
  MatchToolValue( type2, "a" ) returns TRUE
  MatchToolValue( type2, "b" ) returns TRUE
  MatchToolValue( type2, "d" ) returns FALSE
  MatchToolValue( type2, "alb" ) returns FALSE

SEE ALSO
  FindToolType

BUGS
  None

```

```

Icon.library/PutDefDiskObject      Icon.library/PutDefDiskObject

NAME
    PutDefDiskObject - write disk object as the default for its type. (V36)

SYNOPSIS
    status = PutDefDiskObject(diskobj)
    DO
    BOOL PutDefDiskObject(struct DiskObject *);

FUNCTION
    This routine writes out a DiskObject structure, and its
    associated information. If the call fails, a zero will
    be returned. The reason for the failure may be obtained
    via IOErr().

    Note that this function calls PutDiskObject internally which means
    that this call (if successful) notifies workbench than an Icon has
    been created/modified.

    Using this routine protects you from any future changes to
    the way default Icons are stored within the system.

INPUTS
    diskobj -- a pointer to a DiskObject

RESULTS
    status -- TRUE if the call succeeded else FALSE

SEE ALSO
    GetDefDiskObject

BUGS
    None
  
```

```

Icon.library/PutDiskObject          Icon.library/PutDiskObject

NAME
    PutDiskObject - write out a DiskObject to disk.

SYNOPSIS
    status = PutDiskObject(name, diskobj)
    DO
    BOOL PutDiskObject(char *, struct DiskObject *);

FUNCTION
    This routine writes out a DiskObject structure, and its
    associated information. The file name of the info
    file will be the name parameter with a
    ".info" appended to it. If the call fails, a zero will
    be returned. The reason for the failure may be obtained
    via IOErr().

    As of release V2.0, PutDiskObject (if successful) notifies workbench
    that an Icon has been created/modified.

    Using this routine protects you from any future changes to
    the way Icons are stored within the system.

INPUTS
    name -- name of the object (pointer to a character string)
    diskobj -- a pointer to a DiskObject

RESULTS
    status -- TRUE if the call succeeded else FALSE

SEE ALSO
    GetDiskObject, FreeDiskObject

BUGS
    None
  
```




Commodities Exchange

CONTENTS

1. Commodities User Manual	1
1.1 INTRODUCTION	1
1.2 INSTALLATION	1
1.3 CONFIGURATION	1
1.4 EXAMPLES	3
1.5 THE EXCHANGE CONTROLLER	5
2. Commodities Reference Manual	6
2.1 Preface	6
2.2 Introduction	6
2.3 Example Commodities	7
2.4 COMMODITIES COMPONENTS	9
2.5 COMMODITIES OVERVIEW	10
2.6 Examples: Object Structure and Implementation	15
2.7 Examples: More Details	26

1. Commodities User Manual

1.1 INTRODUCTION

Commodities Exchange is a system that makes it easy to write programs which monitor the input handler food chain. This means they can respond to hot keys, take actions based on mouse action or inactivity, or even modify the input stream as it goes by.

Not only are such programs easier to write using Commodities Exchange, but they are managed by a single controller program, Exchange, which provides simple and consistent display and control over the programs.

The components of the system are a support library named "commodities.library", several example commodities, and the special controller program Exchange.

1.2 INSTALLATION

Commodities.library must be present in the LIBS: directory or in the LIBS: path. The icons for the various programs and the Exchange controller can reside on any disk of your choice.

1.3 CONFIGURATION

To start a commodity, you simply double click on its icon. There are several example commodity programs included on the V2.0 release disks. Each commodity has several argument strings that can be passed to it when it is started as follows:

1. If the commodity is started from the Workbench, the ToolTypes strings can be edited via the Info menu item.
2. If the commodity is started from the CLI, you can make a single argument out of each corresponding ToolType syntax. Example:

```
run IHelp "CX_PRIORITY=1" "CYCLE=shift f1" "MAKEBIG=shift f2"
```

Each commodity can support its own specific ToolType arguments. In addition, all commodities support the CX_PRIORITY ToolType:

CX_PRIORITY The CX_PRIORITY ToolType determines when a commodity will see input relative to the other commodities. The commodity with the highest priority will see input before a commodity of lower priority.

The priority is relative to the other commodities only, ie., it does not affect the priority of the task.

For instance, the commodity programs FKey and IHelp let you assign functions to the F1 key. If both FKey and IHelp are running only the one with the highest priority will see the F1 key event.

All commodities that support windows also support the following ToolTypes:

CX_POPKEY The CX_POPKEY ToolType means that this commodity supports a pop-up window, and allows you to specify the hot key (or other input event!) that will make the program show itself. Such programs can be made to disappear either by terminating completely, or going into an invisible ("hidden") state. Convention dictates that the Close gadget on a pop-up window only makes it disappear, not quit. As you will see, you can either quit these programs by another means directly, or via the Exchange controller.

The full syntax description for valid hot key expressions can be found in the Commodities Reference Manual.

CX_POPUP The CX_POPUP ToolType accepts either YES or NO for its input. If YES then the commodity will open its window when first run. If NO the commodity will run but will not open its window. This is useful for putting commodities in the WBStartup drawer.

Note that a commodity started from the CLI can be terminated either by typing Control-E, or by using the BREAK command of the CLI if the commodity was started in the background with the RUN command. Example:

```
BREAK TASK 2 E
```

Commodities started from the Workbench are killed either through an interface of their own, or via the Exchange controller.

Commodities that do not support a window can be killed by running the program a second time either from the WorkBench or the CLI.

1.4 EXAMPLES

There are several commodities included on the 2.0 distribution disks. We'll describe them briefly here.

IHelp An Intuition keyboard enhancer. IHelp monitors input and when triggered by the specified input events it takes several actions corresponding to its ToolTypes.

CYCLE Cycles non-drawer windows back to front.

MAKEBIG Makes a window as large as legally possible without moving it.

MAKESMALL Makes a window its minimum dimensions.

CYCLESCEEN Cycles between all the available screens.

ZIPWINDOW Performs the same function as the zoom gadget in the currently active window.

IHelp has no pop-up window.

NoCapsLock This program is very simple. It renders the caps lock key ineffective. It has no pop-up window.

AutoPoint An example of a "Sun-style" window activation program. When the mouse pointer is moved the window underneath it is automatically activated. It also has no pop-up window.

Blanker This program is a simple screen blanker that blanks the screen after a certain number of seconds of inactivity. It accepts the ToolType SECONDS=xxxx to set the default number of seconds before the screen blanks.

FKey This is a function key assigner which allows you to assign strings to the unshifted and shifted function keys. For the non-shifted function keys it accepts ToolTypes of the form:

```
F1=status full
F2=dir all
.
.
.
F10=list
```

For the shifted function keys, it accepts ToolTypes of the form:

```
SF1=status full
SF2=dir all
.
.
.
SF10=list
```

The window also has four buttons at the bottom that allow you to save the current settings as ToolType strings in the program's icon. The buttons work as follows:

Save	Saves the current settings as ToolType strings in the program's icon and closes the window.
Use	Uses the current settings without saving and closes the window.
Cancel	Restore the settings to those before the window was opened and close the window.
Quit	Terminate FKey without saving the settings.

Blank This program has a pop-up window, but nothing else. It is provided as a skeleton example of a window-based commodity. It responds to all of the Exchange controller commands. It also demonstrates the uniqueness feature: if it is running and you try to start it again, it will just pop-up the window for its existing invocation and the new invocation will abort. This way, if you forget the pop-up hot key for a program, you can get to it either through the Exchange controller (see below) or by clicking on the program's icon again.

This program and its source are available on the DevCon disks.

LeftyMouse Swaps the function of the left and right mouse buttons so southpaws can use the mouse on the left side of the keyboard.

This program and its source are available on the DevCon disks.

MBA Middle Button Action. This commodity converts a middle mouse button event into a shift left button event. Useful if you have a three button mouse.

This program and its source are available on the DevCon disks.

1.5 THE EXCHANGE CONTROLLER

The Exchange program is actually just another commodity, in that it has a pop-up window and is a commodities.library client like the other examples. However, it allows you to view and control the other loaded commodities.

When started, it presents a window with two button panels, a status area, and a scrolling list of loaded commodities (including itself).

The buttons on the left side control the Exchange itself. You can select Hide to make it disappear. (The default CX_POPKEY hot key for Exchange is "alt help".) You can select Quit to terminate the Exchange controller.

The display of commodities is operated with the mouse. Selecting an item will cause the status area to display more detailed information about

Title	Shows the name of the selected commodity.
Description	Gives a brief description of the commodity.
Status	Shows whether the program is enabled or disabled.

Once an item is selected, the buttons on the right hand side apply to that item. (Yes, it's OK to operate on Exchange itself in this way!)

The buttons on the right apply to the selected commodity as described below. Note that the window keys Show and Hide will be ghosted for commodities that do not support windows. If no commodity is selected, all the buttons on the right will be ghosted.

Show	Causes a commodity to pop-up its window. If closed, it will be opened. If open, it will be brought to the front.
Hide	Commands a commodity to close its pop-up window, but not exit.
Disable	Commodities support this command to turn off their input filtering. This is useful to temporarily disable a commodity without actually terminating the program.
Enable	The opposite of the above. Enables a currently disabled commodity
Kill	This causes the selected commodity to terminate. All commodities support this command, either through the Exchange controller or by the issuing of a Control-E break signal directly to their task.

2. Commodities Reference Manual

2.1 Preface

In addition to this document, required reading includes the collected function references, the application include files, and the example applications provided. Familiarity with the input.device, especially the include file <devices/inpotevent.h> is necessary. Exec Lists, Messages and Message Ports play a fundamental role in the system, and must also be understood.

2.2 Introduction

Commodities Exchange has as its major component an Amiga Exec library, named "commodities.library." If this library is put in logical volume LIBS:, it can be opened and used by application programs to gain access to input in a very flexible way. In particular, a Commodities application can monitor the Amiga input stream coming from the input.device without using an Intuition window or a console.

To understand the purpose of Commodities Exchange takes a little explanation. The Amiga computer provides a low-overhead, message-based multi-tasking operating environment, complete with a multi-threaded DOS and Graphics/Windows environment. No other PC provides this, yet the impact sometimes seems to be lost on the personal computing community at large.

To some degree, this is explained by surrogate multi-tasking providing a high degree of satisfaction on popular competitors such as the Apple Macintosh and IBM PC. Small programs called Desk Accessories, Pop-Up programs, Resident programs, Terminate-and-Stay-Resident programs and the like provide multiple application capability, with some limits.

Technical problems with the ad-hoc methods of providing these functions on existing microcomputers find their way to the user, who must often empirically determine what programs may safely coexist, and in what order they must be loaded.

Most of the technical problems in writing such programs are non-existent on the Amiga. In particular, using disk operating system services in pop-up programs is a black art on an IBM PC. A principle area of difficulty or awkwardness is the monitoring of input, be it for the purpose of triggering a pop-up program, performing spelling checking as words are typed, or translating keystrokes into sequences.

On the Amiga, the input problem is fundamentally solved by the input.device which provides an endorsed means of adding "input handlers." Unfortunately, without standards, arbitrary use of input handlers by uncooperative programs can quickly lead to incompatibilities or load-order dependence even on the Amiga. Furthermore, writing input handlers is far from trivial.

Which brings us to the goal of the Commodities Exchange. Commodities consists of a single input handler which precedes the Intuition input handler. Like Intuition, Commodities will route input events to multiple applications. Commodities applications typically are interested in input regardless of what window is active, and thus Commodities is used as a simple, standardized way of managing the input for Resident type programs.

Please follow the standards described in this document for user-specification of the input which triggers an action, and for establishing the priority of individual Commodities applications. In this way many high quality, mutually co-existent "resident" programs can be created by the amazing Amiga programming community.

2.3 Example Commodities

This section describes the end-user's view of several example Commodities applications. The source code for some of these is provided with the release. These programs together illustrate the most common principles of Commodities, and will serve as examples for the rest of this manual.

To run any of the following examples, the user must have a copy of the file "commodities.library" in their LIBS: directory, where it will be found when the program begins execution. All the example applications would normally be started from the Workbench by opening (double-clicking) their icon. The ToolTypes information in the icon is used to configure the programs.

IHelp

The purpose of IHelp is to provide keyboard control of Intuition window operations. This reduces the need for using the mouse while doing keyboard intensive work, such as writing software or documents. This program was the prime motivation for the Commodities Exchange. Managing a multi-tasking user interface is too important a task to assign exclusively to a low-bandwidth device such as a mouse.

Currently, IHelp provides five functions:

CYCLE	This function causes the rearmost application window on the Workbench to be brought to the front and Activated.
MAKESMALL	The active window is shrunk to its minimum dimensions.
MAKEBIG	The active window is made the maximum size possible respecting its limits and the edge of the screen.
CYCLESSCREEN	Cycles through the available screens.
ZIPWINDOW	Same as using a windows zoom gadget.

Each of these functions is assigned to an input event, typically a keystroke such as a function key. Pressing the proper key causes the action to take place regardless of which window is active or what use the active window may be making of keyboard input.

In order to allow the user the ability to associate keystrokes (or other events) with these actions, the ToolType fields in the programs icon are used. The user selects (but does not open) the icon for IHelp, and selects "Info" from the Workbench menu. The subsequent display allows the addition, editing, and deletion of parameter strings called ToolTypes.

To assign the five functions above to new keystrokes, ToolType items are added or edited to read, for example:

```
CYCLE=f1
MAKEBIG=alt f5
MAKESMALL=shift alt f5
CYCLESSCREEN=f4
ZIPWINDOW=f5
```

The icon is saved, and the next time IHelp is loaded these keystrokes prescribed by the user will be associated with the program's actions.

NoCapsLock

This application is almost invisible to the user. It monitors all keystrokes and insures that the CAPSLOCK key on the keyboard never has any effect (regardless of what its little red light says). The shift keys still function normally, but the frequent annoying effects of inadvertent pressing of the CAPSLOCK key are no more.

This program provides nothing the user would want to change, so no ToolType fields in its icon are used.

LeftyMouse

This commodity swaps the function of the left and right mouse buttons. This is for leftys who wish to use the mouse on the left side of the keyboard. Again, no ToolType fields are used.

Blanker

This program provides a simple screen blanker that will blank the screen after a certain number of seconds of inactivity. It accepts the SECONDS=xxxx ToolType to adjust the default timing.

MBA

This program "Middle Button Action" converts a middle mouse button event (not normally used) into a shift left button event for extended icon selection. For use on systems with three button mice.

Blank

Blank is provided as an example of a simple window based commodity. Developers wishing to create their own commodities application should start with this example source code.

2.4 COMMODITIES COMPONENTS

The Commodities Exchange consists of five components: the commodities.library, the Exchange program, a scanned library, include files and example programs. These are described below.

Commodities.library

This is an Amiga Exec library which creates an input handler when it is initialized. It provides a large number of functions to manipulate Commodities Objects and Messages, the building blocks of a Commodities application. All access to data used by this library is done procedurally: no internal data structures are read or modified directly by application programs.

The functions provided by the library fall into these classes:

- Object Creation/Deletion
- Object Linking
- Object Data Access/Modify
- Object Type-specific Functions
- Input Event Matching Utilities
- Message Routing/Disposing

The library must be opened (using the Exec function OpenLibrary()) by each Commodities application.

All parameters to the library, following Amiga library convention, are thirty-two bit quantities. Small-integer users must remember that integers and characters are not promoted to thirty-two bits by the compiler, so casts must be used. In particular, don't forget to cast boolean and priority values to (LONG).

Exchange Controller Program

The Exchange Controller program can be used to see the loaded commodities and perform simple operations such as window show/hide or commodity enable/disable. It is also possible to use this program to terminate an application.

Support Scanned Library

A scanned library is also provided. It is included in amiga.lib. A scanned library contains functions which are linked into the executable file of an application. Included are interface routines to commodities.library and some utility functions.

Include Files

Various include files are provided. For applications, the file <libraries/commodities.h> contains the principle information. Assembler versions of the include files do not exist.

Source Examples

As mentioned, the source for the programs Blank, Blanker, Leftymouse, MBA, IHelp and NoCapsLock is provided on the DevCon disks.

Commodities Applications

You write these. They open "commodities.library" and perhaps use functions in the scanned library. They create and interconnect objects (see below) in such a way that particular events cause desired actions or notifications.

2.5 COMMODITIES OVERVIEW

Objects and Messages

The basic building block is the Commodities Exchange Object, or CxObj. Each object is of some particular 'type' which corresponds to the primitive action that occurs when a Commodities Message arrives at the object. The objects are connected together in a large tree structure and messages corresponding to input events rattle down different paths in the tree, triggering different actions as they encounter different objects.

One particular object type is the Custom object, which calls an application-provided function when a CxMsg arrives, so that programmers can provide virtually any function not available with the standard CxObj types.

A Commodities Exchange Message, or CxMsg, corresponds with few exceptions to a single Input Event. When such a message arrives at a CxObj, an action takes place. This action might be the sending of an Exec message to an application message port, the diversion of the CxMsg down a sub-tree, or the calling of an application-provided function.

All objects can be disabled (inactivated), which inhibits any action being taken when a CxMsg arrives. The message simply proceeds along to its default next destination.

Each CxObj is itself an Exec Node; the objects are linked together in linear lists. Each object also contains a List Header, which we refer to as its *personal list*, so branching off from any object can be another list. So although the entire linked structure of CxObj's is indeed a tree, there is a definite preferred--or default--direction at each node of the tree.

Starting with a Master List of CxObj's, CxMsg's visit every object in a list in turn, but certain objects can divert them so they head off down another list, normally the personal list of the CxObj performing the diversion.

When (and if) the CxMsg reaches the end of a list, it proceeds from where it was diverted. A stack is maintained for each CxMsg to record and recover from multiple diversions.

Brokers and Application Sub-Trees

The CxObj's in the Master List are a special lot. They are called Commodities Brokers and are typically in one-to-one correspondence with application programs. They are "representatives" of the application programs; the sub-tree starting with the personal list of a Broker consists of the CxObj's created by the application. When it is enabled, a Broker will divert *all* CxMsg's it receives down its personal list, into the network of objects the application has set up.

Support is provided for preventing the creation of duplicate brokers. This gives the application an easy way of determining during its initialization if there is another copy of itself already running. The new copy can terminate, and it can be arranged so that the existing copy be notified that a restart attempt was made. A resident but dormant pop-up notepad program might take such a notification that it should open up a window and swing into action.

Standard Objects

The actions performed by CxObj's of the standard types are intended to be primitive. An application wishing to receive an Exec message when a particular keystroke occurs must create 3 CxObj's for the task: one to filter the CxMsg's corresponding to the keystroke, and attached to its list, a CxObj to send off an Exec Message, and another (optional) to swallow the CxMsg.

These three are in addition to the application's (single) broker.

A filter acts by diverting a select set of CxMsg's down its personal list, where they will encounter the other two objects who unconditionally perform their action.

Here is a summary of the CxObj types and a brief description of each:

Broker	Only CxObj's on Master List. Contain a description of the application. Diverts everything down its personal list.
Filter	If CxMsg matches some expression, divert
TypeFilter	If CxMsg has type in specified set, divert
Signal	Signals some task when any CxMsg arrives
Sender	Sends copy of CxMsg to some port (asynchronous)
Translate	Replaces CxMsg with a chain of zero or more new input event CxMsg's
Debug	Dumps message contents to debug device (kprintf)
Custom	Calls a programmer-provided function (synchronous)

There is a more detailed description in the AutoDocs for the functions which create the various types of CxObj.

Custom Objects

Custom objects take a unique action. They call a function provided by the application *synchronously* with the execution of the Commodities handler. The delicacy of this cannot be over-emphasized. The function will execute as part of the input.device task. No DOS or Intuition functions may be called. No assumptions can be made about the values of registers upon entry.

All such functions should be kept quick and simple, with a *minimum* of stack usage. A Custom CxObj is the only way to directly modify input events as their CxMsg's proceed through the Commodities network. Uses other than that should be thought about long and hard and again.

Input Events and Matching Conditions

As mentioned above, most CxMsgs correspond to InputEvents. Now is a good time to reread the Chapter, The Input Device, in the *ROM Kernel Reference Manual*. See also the include file <devices/inputevent.h>.

Each input event which reaches the Commodities handler is sent through the CxObj network as a CxMsg. If it makes it all the way through (i.e., to the end of the Master List) the contents--which may have been changed by some object-- are sent along to subsequent input handlers, including Intuition, and along to window (i.e., non-Commodities) applications.

Flow of an input event CxMsg is determined by its interactions with filter objects. Each filter object has a "matching condition" assigned by the application which determines which messages will be diverted down the filter's personal list (those that match).

The internal representation of this matching condition is private, but can be specified by two mechanisms. Enhancements and additions to these mechanisms can be expected; provisions for upward compatibility are made for both.

Input Description Strings

These are the character strings found in the ToolTypes fields described in the introduction to the sample program IHelp, section 2.3.

Each string describes a subset of input events.

```
[class] [[[-](qual | syn)]] [[-]upstroke] highmap|ansicode
```

The class, qual, syn, and upstroke arguments are optional. The legal values for the arguments are lowercase strings as shown below:

class	One of the strings: rawkey, rawmouse, event, pointerpos, timer, newprefs, diskremoved, diskinserted. If not specified, the class is taken to be rawkey.
qual	Zero or more of the strings: shift, rshift, capslock, control, lalt, ralt, lcommand, rcommand, numericpad, repeat, midbutton, rbutton, leftbutton, relativemouse, A preceding '-' means that the value of the corresponding qualifier is to be considered irrelevant.
syn	One of the strings: shift, caps, alt shift (means "left or right shift"), caps (means "either shift or capslock"), alt (means "either alt key").
upstroke	literally "upstroke" If this token is absent, only downstrokes are considered for rawmouse (mousebuttons) and rawkey events. If it is present alone, only upstrokes count. If it preceded by '-' it means that both up and down strokes are included.
highmap	One of the strings: space, backspace, tab, enter, return, esc, del, up, down, right, left, help, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10.

ansicode A single character token is interpreted as a character code,
which is looked up in the system default keymap.

Examples of strings are:

```
"lshift right alt f2"  
"alt shift a"  
"-shift -alt -control help" (help key with or without qualifiers)  
"rawmouse rbutton"          (mouse move with menu button down)
```

Enhancements to the grammar are anticipated. Every attempt will be made to be compatible with the input descriptions of the earlier versions.

Input Expressions

Applications can also use a powerful binary specification of the matching condition for a filter. In the future, perhaps a more powerful specification may be developed so the data structures for the Input Expressions (type IX) are tagged with their version number. This will allow easy handling of different specification types co-existing in a future Commodities version.

The current Version 2 IX structure is as follows:

```
typedef struct {  
    UBYTE    ix_Version;  
    UBYTE    ix_Class;  
  
    UWORD    ix_Code;  
    UWORD    ix_CodeMask;  
  
    UWORD    ix_Qualifier;  
    UWORD    ix_QualMask;  
    UWORD    ix_QualSame;  
} IX;
```

Here are the matching conditions that each IX field supports. It specifies a match with an InputEvent.

ix_Version Must be set to the manifest constant IX_VERSION found in the same include file
 <libraries/commodities.h> as the structure definition.

ix_Class Must exactly equal the ie_Class field of an InputEvent to match.

ix_Code

ix_CodeMask For every bit set in ix_CodeMask, the state of the corresponding bits in ix_Code and ie_Code in the InputEvent must match. Note that whether a keystroke or mouse button action is a down- or upstroke is encoded in the ie_Code field of the InputEvent (IE_CODE_UPPREFIX).

ix_Qualifier

ix_QualMask

ix_QualSame The fields ix_QualMask and ix_QualMask work in the same way as for the ix_Code: the Mask indicates the "do care" bits in ie_Qualifier, with ix_Qualifier specifying the required settings for those bits. The ix_QualSame field is used to express simple synonyms in the ie_Qualifier. Using this, you can specify that left and right Shift are equivalent and whether the CAPSLOCK qualifier is also equivalent. Left- and right-Alt can also be made equivalent.

Message Routing

So far a few points have come out about routing CxMsg's:

- In the absence of some action diverting them, after visiting a CxObj, they will proceed to the next node in the same list.
- They may be diverted down the "personal list" of some CxObj, normally the list belonging to the object doing the diverting. Diverting consists of pushing the address of the current CxObj onto an internal stack and routing to the first CxObj in the list.
- When they reach the end of a list of CxObj's, the address of a CxObj is popped off of their stack, and they proceed to the Successor of that object.
- Reaching the end of the Master List sends the CxMsg to an internal "Zero Object" which disposes of the message, first recovering InputEvents from the contents of input event messages.

Application programs (using Custom Objects) can route a message two ways: diverting it down the list of some object, or by routing it directly to another object. Future functions may be provided for independently pushing or popping the routing stack.

Note that it makes no sense to try to route a message sent to a port by a sender CxObj.

2.6 Examples: Object Structure and Implementation

With the background so far, it is helpful to consider the object structure used by the example programs discussed above.

Common

The code in the Common directory on the DevCon disks is shared by all the commodities. The directories for the individual commodities contain only two files, app.c and app.h. These files contain the application specific code only. A commodity is formed by linking the application specific app.c with the files in the Common directory. By sharing all non-application specific code amongst the commodities it is easier to maintain a consistent, standard user interface. Sharing of the code also facilitates easier maintenance of the non-application specific code. The Common directory contains the following files:

- main.c** This file contains the main program code as well as the program termination code. The code in the file main.c is responsible for the following:
- Opening the required libraries.
 - Allocating the custom signal if required.
 - Parsing the ToolTypes or command line.
 - Initializing the commodities specific code by calling routines in cx.c.
 - Opening the window (if supported) by calling routines in window.c.
 - Waiting for the CTRL_E termination signal.
 - Waiting for and dispatching all incoming messages both from Intuition and the commodities.library.
- cx.c** This file contains all the commodities specific setup and shutdown code. This file is responsible for the following:
- Setting up the message port for commodities messages.
 - Setting up the commodities broker.
 - Setting up the hot key if a window is supported.
 - Calling the application specific commodities setup code.
 - Checking the accumulated commodities error codes.
 - Activating the commodities broker.

- Waiting for and handling or dispatching commodities messages including:
 - The message for the applications CX_POPKEY which generates a call to window.c to open the pop-up window.
 - A message destined for the application specific code which generates a call to handleCustomCXMsg() in the applications app.c module.
 - A CXM_COMMAND message such as CXCMD_DISABLE, CXCMD_ENABLE, CXCMD_UNIQUE, CXCMD_APPEAR, CXCMD_DISAPPEAR, CXCMD_KILL, or a custom command.
- Shutting down the commodities specific code by:
 - Shutting down the applications custom commodities code in app.c.
 - Deleting all the commodities objects.
 - Clearing the commodities message port.

window.c

This file contains all the window manipulation code. This file is responsible for the following:

- Setting up the window by:
 - Doing a WindowToFront() if the window is already open or opening the window if it is not already open.
 - Setting up the windows message port.
 - Using GadTools.library to add the gadgets and menus to the window according to the application supplied routines in app.c.
- Handling and dispatching Intuition messages such as CLOSEWINDOW, REFRESHWINDOW, MENU PICK, and GADGETUP. Application specific messages are dispatched to their handlers in app.c.
- Shutting down the window by:
 - Saving the windows current position and size.
 - Clearing the menus.
 - Closing the window.
 - Removing the gadgets.

IHelp

IHelp has three hot keys. There is a routine in the support library, `HotKey()`, that creates these groups of objects easily.

A hot key triad, to review, consists of a filter object, which is the "trigger" for the hot key. Attached to its personal list is a sender which sends a notification message to a port allocated by IHelp.

The ID field of the several senders differ, so that IHelp can easily tell from which sender the messages were sent. IHelp pays no attention to the data part of the messages it is sent.

Following the sender in the personal list of the filter is a translator object which translates the trigger `CxMsg` to `NULL`, in effect swallowing it.

IHelp, like all good commodities, has a Broker, and the filter objects at the head of the hot keys are all attached to its list.

The commodities specific setup code installs the programs five hot keys as follows:

```
BOOL setupIHelp()
{
    LONG          error;

    AttachCxObj(broker,
        HotKey( ArgString(ttypes, "CYCLE", "f1"), cxport, CYCLE) );
    AttachCxObj(broker,
        HotKey( ArgString(ttypes, "MAKEBIG", "f2"),  cxport, MAKEBIG) );
    AttachCxObj(broker,
        HotKey( ArgString(ttypes, "MAKESMALL", "f3"), cxport, MAKESMALL) );
    AttachCxObj(broker,
        HotKey( ArgString(ttypes, "CYCLESCREEN", "f4"), cxport, CYCLESCREEN) );
    AttachCxObj(broker,
        HotKey( ArgString(ttypes, "ZIPWINDOW", "f5"), cxport, ZIPWINDOW) );

    if (error = CxObjError(broker))
    {
        D( printf("accumulated broker error %ld0, error) );
        return(0);
    }
    return(1);
}
```

The program then receives commodities messages at export and dispatches them as follows:

```
VOID MyHandleCXMsg(id)
ULONG id;
{
    switch(id)
    {
        case CYCLE:
            D( printf("cycleforward0) ");
            cycleforward();
            break;
        case MAKEBIG:
            D( printf("makebig0) ");
            makesize((int) MAKEBIG);
            break;
        case MAKESMALL:
            D( printf("makesmall0) ");
            makesize((int) MAKESMALL);
            break;
        case CYCLESCREEN:
            cyclescreen();
            break;
        case ZIPWINDOW:
            zipwindow();
            break;
    }
}
```

NoCapsLock

Attached to the Broker list of this application is a single filter which watches for all RAWKEY events which have the CAPSLOCK bit set. The setup code for the filter follows:

```
/* An input expression to match any RAWKEY event with the
   CAPSLOCK qualifier bit set */

IX myix = {
    IX_VERSION,          /* required */
    IECLASS_RAWKEY,
    0,                   /* Code: won't care */
    0,                   /* CodeMask: 0 means don't care about Code */
    IEQUALIFIER_CAPSLOCK, /* qualifier I am interested in */
    IEQUALIFIER_CAPSLOCK, /* and it's the only qualifier of interest */
    0                    /* synonyms irrelevant */
};

BOOL setupNoCapsLock()
{
    CxObj  *filter;

    filter = CxFilter(NULL);
    if (!filter)
        return(0);

    SetFilterIX(filter, &myix);
    AttachCxObj(filter, CxCustom( nocapsaction, 0L));

    if (CxObjError(filter))
    {
        D( printf("nocapslock: filter error %lx0, CxObjError(filter) ) );
        DeleteCxObjAll(filter);
        return(0);
    }

    AttachCxObj(broker, filter);
    return(TRUE);
}
```

The filter diverts all matching messages down its list where a single custom object can be found. The code for the custom object follows:

```
void nocapsaction(cxm,co)
register struct CxMsg *cxm;
CxObj *co;
{
    struct InputEvent *ie;

    D( kprintf("nocapsaction0) );

    /* i KNOW that all messages getting this far are CXM_IEVENT */
    ie = (struct InputEvent *) CxMsgData(cxm);

    ie->ie_Qualifier &= ~IEQUALIFIER_CAPSLOCK;
}
```

This custom object modifies the InputEvent included in each CxMsg it encounters, clearing the CAPSLOCK bit. It has no effect on other Qualifier bits, so even if the CapsLock key is active, the shift keys will function normally.

MBA

Attached to the Broker list of this application is a single filter which watches for all RAWMOUSE events which have the MBUTTON bit set. The setup code for the filter follows:

```
/* An input expression to match any RAWMOUSE event with the
   MBUTTON qualifier bit set */

IX middleix = {
    IX_VERSION,          /* required */
    IECLASS_RAWMOUSE,
    IECODE_MBUTTON,      /* Code */
    ~IECODE_UP_PREFIX,   /* CodeMask */
    0,                   /* qualifier I am interested in */
    0,
    0                     /* synonyms irrelevant */
};

BOOL setupMBA()
{
    CxObj *mfilter;

    mfilter = CxFilter(NULL);
    if (!mfilter)
        return(0);

    SetFilterIX(mfilter, &middleix);
    AttachCxObj(mfilter, CxCustom( middlebaction, 0L));

    if (CxObjError(mfilter))
    {
        D( printf("MBA: filter error %lx0, CxObjError(mfilter) ) );
        DeleteCxObjAll(mfilter);
        return(0);
    }
    AttachCxObj(broker, mfilter);

    return(TRUE);
}
```

This custom object modifies the InputEvent included in each CxMsg it encounters, setting the event to a shift left button event as follows:

```
void middlebaction(cxm,co)
register struct CxMsg *cxm;
CxObj *co;
{
    struct InputEvent *ie;

    ie = (struct InputEvent *) CxMsgData(cxm);

    /* i KNOW that all messages getting this far are CXM_IEVENT */
    /* convert middle button into shift left */
    ie->ie_Code = (ie->ie_Code & IECODE_UP_PREFIX) | IECODE_LBUTTON;
    ie->ie_Qualifier |= IEQUALIFIER_LSHIFT;
}
```

Blanker

Blanker has a single custom object attached to its broker. This custom object checks every message that goes by. If a certain number of timer events (which happen every 10th of a second) go by with no other activity the screen is blanked. Once blanked, any event except a timer event will cause the screen to unblank. The setup code for blanker follows:

```
BOOL setupBlanker(VOID)
{
    CxObj    *objectlist;

    seconds = ArgInt( ttypes, "SECONDS", 60 );

    objectlist=CxCustom( BlankerAction, 0L);

    if (CxObjError(objectlist))
    {
        D( printf("blanker: filter error %lx0, CxObjError(objectlist) ) );
        DeleteCxObjAll(objectlist);
        return(0);
    }

    AttachCxObj(broker, objectlist);
    return(TRUE);
}
```

The code for the custom action takes advantage of the custom signal as follows:

```
VOID BlankerAction(struct CxMsg *cxm,CxObj *co)
{
    register struct InputEvent *ie;

    ie = (struct InputEvent *) CxMsgData(cxm);

    if(ie->ie_Class==IECLASS_TIMER)
    {
        if(! blanked)
        {
            /*D( printf("blanked=0x%lx Bump timeout=%ld0,blanked,timeout);)
            if(++timeout >= ((ULONG)(seconds*10L)))
            {
                D( printf("#####Blank Screen0); )
                BlankerCommand=BLANK;
                Signal(maintask,csigflag);
                blanked=TRUE;
            }
        }
    } else {
        if(ie->ie_Class!=IECLASS_TIMER)
        {
            D( printf("#####UnBlank0); )
            BlankerCommand=UNBLANK;
            Signal(maintask,csigflag);
            timeout=0L;
            blanked=FALSE;
        }
    }
}
```

When using a custom signal the application must supply a routine in app.c to handle the custom signal as follows:

```
VOID MyHandleCustomSignal (VOID)
{
    D( kprintf("custom: MyHandleCustomSignal() enter0); )
    if (BlankerCommand==BLANK)
        Blank();
    else
        if (BlankerCommand==UNBLANK)
            UnBlank();
}
```

2.7 Examples: More Details

Error Handling

A well-behaved program must always check that functions which create CxObj's succeed, since each object requires some dynamically allocated memory. To make this a little less tedious, it is possible to get by testing if an entire list of objects was created successfully, and to recover gracefully if not. See the examples. And check for and handle all errors, OK?

ToolTypes and the Commodities Environment

Some functions are provided to make it easy to use ToolTypes for providing user control over the parameters of your application. Please let the user specify the input description for major semantic actions in your programs.

Also, even if there is no obvious reason in some cases, let the user set the priority for your broker. That is the key step to making the functioning of a group of Commodities applications independent of the order in which they were loaded.

Note that the functions provided also support command line argument parsing so commodities can be started from the CLI with full user control (this is especially useful for starting up a batch of applications from an Execute script file).

Library version

Be sure to specify a non-zero version number to the OpenLibrary() call when opening Commodities. This insures that the end user does not use your program with a version of the library that doesn't support new features you are depending on.

Terminating a Commodities Application

As is shown in the examples, the proper terminating condition of a Commodities application is the receipt of the Control-E break signal. This signal will be sent by controller software to the task which created a Broker when the user indicates that that Broker's program is to be terminated. It is also easy to send such a signal to a background task started from the CLI. Use the "Status" command to find the tasks CLI ID, and then say "Break task <n> E" where <n> is the number from the Status output.

Developing a Commodities application is made more convenient using this convention (we often test them in the foreground, and typing Control-E usually works long before the rest of the program), and we also don't waste function keys or other input combinations on termination commands.

Pools and Alerts

The internal data structures used by the commodities.library are allocated and managed by a common "pool" mechanism. Normally, this mechanism is invisible to the programmer and end-user. It is used to manage data storage for Commodities objects, messages, Exec messages, and InputEvents.

A recoverable alert will be put up if a program corrupts the secret header of a Commodities data structure, or uses an invalid handle in certain operations. Another recoverable alert is posted when commodities.library expunges (which is done when the last application closes the library) if all data structures are not returned to commodities.library before this time. The frequent causes for this are not replying to all the Exec messages Commodities senders send to you, and not deleting all the objects that you have created.

To properly reply all Exec messages sent to your program, a sequence like that used in IHelp is advised:

```
DeleteCxObjAll(broker);  
while (msg = GetMsg(port)) ReplyMsg(msg);  
CloseLibrary(CxBase);
```

```

/*****
 *
 *      COPYRIGHTS
 *
 *      UNLESS OTHERWISE NOTED, ALL FILES ARE
 *      Copyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved.
 *
 *****/

/* local.h - Definitions used by standard modules */

#ifndef LATTICE
#include <clib/all_protos.h>
#include <clib.h>
#include <string.h>
#endif

#include "app.h"

/*****
 *
 *      main.c
 *
 *****/

extern struct IntuitionBase *IntuitionBase;
extern struct Library *Cbase;
extern struct GfxBase *GfxBase;
extern struct DosLibrary *DosLibrary;
extern struct Library *GadtoolBase;

extern struct MsgPort *csrcport; /* commodities messages here */
extern ULONG csrcflag; /* signal for above */

extern struct MsgPort *lport; /* Intuition IDCMP messages here */
extern ULONG lsrcflag; /* signal for above */

/*****
 *
 *      CF.C
 *
 *****/

extern char hotkeybuff[]; /* holds the string describing the popup */
/* hotkey. Used for the window title */

VOID handlEcMMsg(struct Message *);
BOOL setupCX(char **);
VOID shutdownCX(VOID);

/*****
 *
 *      window.c
 *
 *****/

VOID handleMMsg(struct IntuiMessage *);
struct Window *getNewWindow(VOID);
int addgadgets(struct Window *);
VOID removegadgets(VOID);

#define PRIORITY_TOOL_TYPE "CX_PRIORITY"
#define POP_ON_START_TOOL_TYPE "CX_POPUP"
#define POPKEY_TOOL_TYPE "CX_POPKEY"

```

```

/* *****
 *
 * COPYRIGHTS
 *
 * UNLESS OTHERWISE NOTED, ALL FILES ARE
 * Copyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved.
 *
 * *****
 */

/*
 * main.c -- Skeleton program for a typical Commodities application.
 */

#include "local.h"

struct IntuitionBase *IntuitionBase = NULL;
struct Library *CkBase = NULL;
struct GfxBase *GfxBase = NULL;
struct DosLibBase *DosBase = NULL;
struct GadToolBase *GadToolBase = NULL;
struct Library *IconBase = NULL;

char **types;

/* these globals are the connection between the main program
 * loop and the two message handling routines
 */

struct MsgPort *cexport = NULL; /* commodities messages here */
ULONG cxsigflag = 0; /* signal for above */
struct MsgPort *limport = NULL; /* Intuition IDCMP messages here */
ULONG lsigflag = 0; /* signal for above */

#if CSIGNAL
LONG csignal = -1L;
struct Task *maintask = NULL;
#endif
ULONG csigflag = 0L;

VOID main(int, char **);
VOID main(argc, argv)
int argc;
char **argv;
{
    ULONG sigrcvd;
    struct Message *msg;
    M(char *str);

    CkBase = (struct DosLibBase *) OpenLibrary("commodities.library", 5);
    DosBase = (struct IntuitionBase *) OpenLibrary("dos.library", 0);
    IntuitionBase = (struct IntuitionBase *) OpenLibrary("intuition.library", 33);
    GfxBase = (struct GfxBase *) OpenLibrary("graphics.library", 0);
    GadToolBase = (struct GadToolBase *) OpenLibrary("gadtools.library", 36);
    IconBase = (struct IconBase *) OpenLibrary("icon.library", 33);

    if ( ! ( IntuitionBase && CkBase && GfxBase && DosBase && GadToolBase && IconBase ) )
    {
        DL( kprintf("main.c: main() Failed to open one or more libraries\n") );
        terminate();
    }

    if CSIGNAL
        if((csignal = AllocSignal(-1)) == -1)

```

```

    {
        DL( kprintf("main.c: main() Failed to get custom signal\n") );
        terminate();
    }
    cxsigflag = 1L << csignal;
    maintask = FindTask(0L);
    #endif

    /* commodities support library function to find argv or tooltypes */
    DL(
        types = ArgArrayInit( argc, argv );
        int xx=0;

        while( types[xx] )
        {
            printf("types[%04x]=%s\n", xx, types[xx]);
            xx++;
        }

        if ( ! setupCX( types ) )
        {
            DL( kprintf("main.c: main() setupCX failed\n") );
            terminate();
        }

        str = ArgString( types, POP_ON_START_TOOL_TYPE, CX_DEFAULT_POP_ON_START );
        if( strcmp(str, "yes") == 0 ) /* will try to setup lport
            setupWindow(); */
            DL( kprintf("main.c: main() After setupWindow\n") );

        for ( ;; ) /* exit by calling terminate */
        {
            /* handling two ports:
             * either will wake us up;
             * simple approach often works.
             */
            DL( kprintf("main.c: main() Waiting for a signal\n") );
            sigrcvd = Wait ( SIGBREAKF_CTRL_E | lsigflag | cxsigflag );
            DL( kprintf("main.c: main() Received a signal\n") );

            /* commodities convention: easy to kill */
            if ( sigrcvd & SIGBREAKF_CTRL_E )
            {
                DL( kprintf("main.c: main() Received a SIGBREAKF_CTRL_E\n") );
                terminate();
            }

            #if CSIGNAL
            if ( sigrcvd & cxsigflag )
            {
                DL( kprintf("main.c: main() Received a Custom Signal\n") );
                handleCustomSignal();
            }
            #endif

            while( cexport && (msg = GetMsg(cexport)) )
                handleCkMsg(msg);
            while( limport && (msg = (struct Message *) GetIMsg(limport)) )
                handleMsg((struct IntuiMessage *) msg);
        }
    }

```



```

/****! Blank.id/terminate() *****/
NAME
    terminate -- Cleanup all resources and exit the program.
SYNOPSIS
    terminate()
    VOID terminate(VOID);
FUNCTION
    This function performs all the necessary cleanup for the
    commodity and calls exit() to return control to the OS.
    No matter how the program exits this should be the last function
    called.
INPUTS
    None.
RESULT
    All resources are freed and the program exits.
EXAMPLE
    If(!AllocWindow())
        terminate();
NOTES
    This function must be set up so that it can be called at any
    time regardless of the current state of the program.
BUGS
SEE ALSO
    shutdownCX();
    shutdownWindow();
*****
*/
VOID terminate()
{
    DI( kprintf("main.c: terminate() enter\n") );
    shutdownCX();
    DI( kprintf("main.c: terminate(), after shutdownCX()\n") );
    W(
        shutdownWindow();
        DI( kprintf("main.c: terminate(), after shutdownWindow()\n") );
    )
    ArgArrayDone(); /* cx_supp.lib function */
    DI( kprintf("main.c: terminate(), after ArgArrayDone()\n") );
    if(CSignal != -1) FreeSignal(CSignal);
    DI( kprintf("main.c: terminate(), after FreeSignal()()\n") );
    #endif
    DI( kprintf("main.c: terminate() CxBase = %i\n",CxBase) );
    DI( kprintf("main.c: terminate() IntuitionBase = %i\n",IntuitionBase) );
    DI( kprintf("main.c: terminate() GfxBase = %i\n",GfxBase) );
    DI( kprintf("main.c: terminate() DOSBase = %i\n",DOSBase) );
    DI( kprintf("main.c: terminate() GadtoolBase = %i\n",GadtoolBase) );
}

```

```

DI( kprintf("main.c: terminate() IconBase = %i\n",IconBase) );
if(CxBase) CloseLibrary(CxBase);
if(IntuitionBase) CloseLibrary((struct Library *)IntuitionBase);
if(GfxBase) CloseLibrary((struct Library *)GfxBase);
if(DOSBase) CloseLibrary((struct Library *)DOSBase);
if(GadtoolBase) CloseLibrary(GadtoolBase);
if(IconBase) CloseLibrary(IconBase);
DI( kprintf("main.c: terminate(), after CloseLibrarys()\n") );
exit(0);
}

```

```

/* *****
 *
 *      COPYRIGHTS
 *
 *      UNLESS OTHERWISE NOTED, ALL FILES ARE
 *      Copyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved.
 *
 *      *****
 */

/*
 * cx.c -- Commodities Interface
 *
 * This module handles all the command messages from commodities.library.
 * Commands such as HIDE SHOW ENABLE DISABLE KILL are sent to commodities
 * from the Exchange program and are processed here.
 */

#include "local.h"

CObj *broker = NULL; /* Our broker */

/* a little global for showing the user the hockey */
char hockeybuff[257];

struct NewBroker mynb = {
    NA_VERSION,
    COM_NAME,
    COM_TITLE,
    COM_DESCR,
    NBU_NOTIFY | NBU_UNIQUE,

    /* Library needs to know version */
    /* broker internal name */
    /* commodity title */
    /* description */
    /* We want to be the only broker */
    /* with this name and we want to */
    /* be notified of any attempts */
    /* to add a commodity with the */
    /* same name */
    /* flags */
    /* default priority */
    /* port, will fill in */
    /* channel (reserved) */
    0,
    0,
    NULL,
    0
};

/* *****
 *
 *      Blank.id/handleCxMsg() *****
 */

NAME handleCxMsg -- Handle Incoming commodities messages.

SYNOPSIS
    handleCxMsg(msg)
    VOID handleCxMsg(Struct Message *msg);

FUNCTION
    This function handles commodities messages sent to the brokers
    message port. This is where the standard commodities features
    such as Enable/Disable Show/Hide Quit and HotKey Popup are
    handled. If the message is not for the standard handler then the
    function handleCustomCxMsg() is called to handle application
    specific events otherwise handleCustomCxCommand() is called
    to handle application specific commodities command messages.

INPUTS
    msg = A commodities message;

RESULT
    Either the standard commodities function is performed or the
    custom handler is called.

```

```

*
*      EXAMPLE
*
*      NOTES
*
*      BUGS
*
*      SEE ALSO
*
*      *****
*/

VOID handleCxMsg(Struct Message *msg)
{
    ULONG msgid;
    ULONG msgtype;

    msgid = CxMsgID(msg);
    msgtype = CxMsgType(msg);

    DI(printf("cx.c: handleCxMsg() enter\n"));

    ReplyMsg(msg);

    switch(msgtype)
    {
        case CXM_EVENT:
            DI(printf("cx.c: handleCxMsg(CXM_EVENT)\n"));
            switch(msgid)
            {
                /*
                 * case POP KEY ID:
                 *     DI(printf("cx.c: handleCxMsg(POP_KEY_ID)\n"));
                 *     setupWindow();
                 *     break;
                 */
                default:
                    DI(printf("cx.c: handleCxMsg(Custom Message)\n"));
                    handleCustomCxMsg(msgid);
                    break;
            }
            break;
        case CXM_COMMAND:
            DI(printf("cx.c: handleCxMsg(CXM_COMMAND)\n"));
            switch(msgid)
            {
                case CXCMD_DISABLE:
                    DI(printf("cx.c: handleCxMsg(CXCMD_DISABLE)\n"));
                    ActivateCObj(broker, 0L);
                    break;
                case CXCMD_ENABLE:
                    DI(printf("cx.c: handleCxMsg(CXCMD_ENABLE)\n"));
                    ActivateCObj(broker, 1L);
                    break;
                case CXCMD_APPEAR: /* Time to pop up the window */
                    /* Someone has tried to run us again */
                    DI(printf("cx.c: handleCxMsg(CXCMD_APPEAR(CXCMD_UNIQUE)\n"));
                    /* If someone tries to run us a second time the second copy
                     * of the program will fail and we will be sent a
                     * CXCMD UNIQUE message. If we support a window then we
                     * Make our window appear since that is what the user wanted.
                     * If we do not support a window then we kill the currently
                     * running version 'this one' so that things like autopoint
                     * can be toggled on/off by running them a second time.
                     */
            }
            break;
    }
}

```

```

        if(WINDOW)
        {
            M( setupWindow() )
        } else {
            terminate();
        }
    }
    break; /* the window
case CXCMD_DISAPPEAR: handleCXMg(CXCMD_DISAPPEAR)\n") )
    M(
        shutdownWindow();
    )
    break;
case CXCMD_KILL:
    D1( printf("cx.c: handleCXMg(CXCMD_KILL)\n") )
    terminate();
    break;
default:
    D1( printf("cx.c: handleCXMg(Custom Command)\n") )
    handleCustomCXCommand(msgid);
    break;
} /* end switch(command) */
} /* end switch(msgtype) */
}

/***1* Blank.I/d/setupCX() *****
NAME
    setupCX -- Initialize commodities.library specific features.
SYNOPSIS
    result = setupCX(ttypes)
    BOOL setupCX(char **ttypes)
FUNCTION
    This function performs all the commodities.library specific
    setup required for a standard commodity. It sets up the brokers
    message port, and priority. And sets certain flags in the
    broker structure so the exchange program will know what
    features this broker supports. If the commodity supports a window
    the window hockey is added to the broker and then the function
    setupCustomCX() is called to setup the application specific
    commodities objects. If all this goes well the broker is activated
    and the function returns TRUE.
INPUTS
    ttypes - NULL terminated Argument array containing this
    applications TOOLTYPES strings.
RESULT
    Returns TRUE if all went OK else returns FALSE.
EXAMPLE
    if(setupCX(ttypes))
    {
        printf("Commodities successfully initialized.\n");
    } else {
        printf("Commodities initialization error!\n");
    }
NOTES
    This function can be called at anytime to reinitialize the
    commodities code from a new set of arguments.
BUGS

```

```

* SEE ALSO
*   setupCustomCX()
*   shutdownCX()
*   shutdownCustomCX()
*
*****
*/
BOOL setupCX(char **ttypes)
{
    LONG error;
    M(char *str)
    D1( printf("cx.c: setupCX() enter\n") )
    shutdownCX(); /* remove what was and create */
                  /* everything from scratch */
    export=CreatePort(myhb.nb_Name,0L) /* Create Message port
    if( ! export )
    {
        D1( printf("cx.c: setupCX() Could not create message port\n") )
        return(FALSE);
    }
    D1( printf("cx.c: setupCX() export=0x%x\n",export) )
    cxsigflag = 1L << EXPORT_OMP_SIGBIT; /* Create signal mask for wait */
    /* Set the brokers priority from the TOOLTYPES or from default if no */
    /* TOOLTYPES are available. Set the brokers Message port. */
    myhb.nb_Pri = Argint( ttypes, PRIORITY_TOOL_TYPE, CX_DEFAULT_PRIORITY );
    myhb.nb_Port = export;
    D1( printf("cx.c: setupCX() myhb.nb_pri=0x%x\n",myhb.nb_Pri) )
    /* If this commodity supports a window then set the SHOW/HIDE flag */
    /* so the Exchange controller can ghost its gadgets appropriately */
    M(myhb.nb_Flags |= COF_SHOW_HIDE);
    /* Attempt to create our broker */
    if( ! ( broker = CxBroker( myhb, NULL ) ) )
    {
        D1( printf("cx.c: setupCX() could not create broker\n") )
        shutdownCX();
        return(FALSE);
    }
    D1( printf("cx.c: setupCX() broker=0x%x\n",broker) )
    /* If this commodity supports a window then add its HotKey now */
    M(
        /* Install a hotkey for popping up window */
        AttachCxbp(broker,
            Hockey[ctr=Argstring(ttypes,POPKEX_TOOL_TYPE,CX_DEFAULT_POP_KEY),
                export,FOR_KEY_ID] );
        strncpy(hotkeybuff,str,sizeof(hotkeybuff)-1);
    )
    /* Setup all application specific commodities objects */
    if( ! setupCustomCX() )
    {
        D1( printf("cx.c: setupCX() setupCustomCX failed\n") )
        shutdownCX();
        return(FALSE);
    }
}

```

```

/* Check for accumulated error */
if ( error = CxObjError( broker ) )
{
    DI( printf("cx.c: setupCX() accumulated broker error %ld\n",error) );
    shutdownCX();
    return (FALSE);
}

/* All went well so activate our broker */
ActivateCxbObj(broker,1L);

DI( printf("cx.c: setupCX() returns TRUE" ) );
return (TRUE);
}

/****1* Blank.Id/shutdownCX() *****/
NAME      shutdownCX -- Cleanup all commodities brokers and handlers.
SYNOPSIS  shutdownCX()
          VOID shutdownCX(VOID);

FUNCTION
Shuts down and cleans up all variables and data used for supporting
the commodities library side of this commodity. This function
must be set up so that it can be called regardless of the current
state of the program. This function handles all the standard
cleanup and calls shutdownCustomCX(); to cleanup the application
specific code.

INPUTS
None.

RESULT
The commodities specific code is cleaned up and made ready for
a terminate(); or a call to setupCX();.

EXAMPLE

NOTES
The first thing that setupCX() does is call this routine. Therefore
this function must work even before your commodity has been
initialized.

BUGS

SEE ALSO
setupCX();
setupCustomCX();
shutdownCustomCX();

*****
*/
VOID shutdownCX()
{
    struct Message *msg;

    DI( printf("cx.c: shutdownCX() enter broker now: %ld\n", broker ) );
    shutdownCustomCX();
    if(!cxport)
    {

```

```

    DI( printf("cx.c: shutdownCX() Deleting all objects\n" ) );
    DeleteCxbObjAll(broker); /* safe, even if NULL */

    /* now that messages are shut off, clear port */
    while(msg=GetMsg(cxport)) ReplyMsg(msg);
    DeletePort(cxport);

    cxport = NULL;
    cxsigflag = 0;
    broker = NULL;
    }
    DI( printf("cx.c: shutdownCX() returns\n" ) );
}

```

```

/*****
 *
 *      COPYRIGHTS
 *
 *      UNLESS OTHERWISE NOTED, ALL FILES ARE
 *      Copyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved.
 *
 *****/

/* window.c -- Intuition Interface */

#include "local.h"

static BYTE dummy;

/* if WINDOW */
/*##### All the following is disabled if the commodity #####*/
/*##### does not support a window. #####*/

struct Window *window = NULL; /* our window */

void
struct Screen *myscreen = NULL;
SHORT
struct TextFont *font = NULL;
struct Gadget *glist = NULL;
struct Menu *menu = NULL;
BOOL
struct DrawInfo *mydi = NULL;
BOOL
IDCMPRefresh = NULL;

/* save window positions and dims left,top,width,height */
static WORD saveWindow[4] = {WINDOW_LEFT, WINDOW_TOP, WINDOW_WIDTH, WINDOW_HEIGHT};

static char WindowTitle[255]; /* buffer to hold cooked window title */

/****! Blank.Id/handleMsg() *****/

NAME
handleMsg -- Handle window IDCMP messages.

SYNOPSIS
handleMsg(msg);

VOID
handleMsg(struct IntuiMessage *msg);

FUNCTION
This function handles all the IntuiMessages for standard
commodities functions. If the message is for an application
Gadget or Menu the appropriate application function,
handleCustomMenu() or handleGadget(), is called.

INPUTS
msg = The current IntuiMessage.

RESULT
The appropriate action for the message is performed.

EXAMPLE

NOTES

BUGS

SEE ALSO

```

```

*
*****/
VOID handleMsg(struct IntuiMessage *msg)
{
    ULONG class;
    UMORD code;
    struct Gadget *gaddress;

    class = msg->Class;
    code = msg->Code;
    gaddress = (struct Gadget *) msg->IAddress;

    DI( kprintf("window.c: handleMsg() enter\n") );

    GT_ReplyMsg( (struct Msg *) msg );

    switch ( class )
    {
        case CLOSEWINDOW:
            DI( printf("window.c: handleMsg(CLOSEWINDOW)\n") );
            shutdownWindow(); /* not reached */
            break;
        case REFRESHWINDOW:
            DI( printf("window.c: handleMsg(REFRESHWINDOW)\n") );
            IDCMPRefresh = TRUE;
            refreshWindow();
            IDCMPRefresh = FALSE;
            break;
        case MENUPICK:
            DI( printf("window.c: handleMsg(MENUPICK)\n") );
            handleCustomMenu(code);
            break;
        case GADGETUP:
            DI( printf("window.c: handleMsg(GADGETUP) GadgetID=%i\n", gaddress->GadgetID) );
            handleGadget(gaddress->GadgetID & GADTOOLMASK, code);
            break;
    }

    /****! Blank.Id/setupWindow() *****/

NAME
setupWindow -- Perform whatever steps necessary to make the
window visible.

SYNOPSIS
setupWindow();

VOID
setupWindow(VOID);

FUNCTION
This function is used to make the window visible. If the window
is not opened this function will open it. If the window is already
open it will be brought to the front so it is visible. This
routine handles all the ugliness of new look and changing window
title bar font heights.

INPUTS

RESULT

EXAMPLE

NOTES

```

```

*   BUGS
*
*   SEE ALSO
*
* .....
*/
VOID setupWindow()
{
    DI( printf("window.c: setupWindow() enter\n") );

    if(window)
    {
        DI( printf("window.c: setupWindow() WindowToFront()\n") );
        WindowToFront( window );
        return; /* already setup, nothing to re-init */
    }
    if( ! myscreen )
    {
        if( ! (myscreen=LockPubScreen(NULL)) )
        {
            DI( printf("window.c: setupWindow() could not LockPubScreen()\n") );
            return;
        }
        DI( printf("window.c: setupWindow() LockPubScreen(NULL) = %lx\n",myscreen) );
    }
    if( ! myd1 )
    {
        if( ! (myd1=getScreenDrawInfo(myscreen)) )
        {
            DI( printf("window.c: setupWindow() could not GetScreenDrawInfo()\n") );
            return;
        }
    }
    DI( printf("window.c: setupWindow() GetScreenDrawInfo(0x%x) = %lx\n",myscreen,myd1) );
    DI( printf("window.c: setupWindow() topborder = %ld\n",topborder) );
    if( ! v1 )
    {
        if( ! (v1=GetVisualInfo(myscreen,TAG_DONE)) )
        {
            DI( printf("window.c: setupWindow() could not GetVisualInfo()\n") );
            goto EXIT;
        }
    }
    DI( printf("window.c: setupWindow() GetVisualInfo() = %lx\n",v1) );
    if( ! (window=getNewWindow()) )
    {
        DI( printf("window.c: setupWindow() could not getNewWindow()\n") );
        goto EXIT;
    }
    DI( printf("window.c: setupWindow() getNewWindow() = %lx\n",window) );
    lport = window->UserPort;
    lsigflag = 1L << lport->mp_sigbit;
    addGadgets(window);
    setupCustomMenu();
    if(menu)

```

```

{
    if( ! LayoutMenus(menu,v1,TAG_DONE) )
    {
        DI( printf("window.c: setupWindow() could not LayoutMenus()\n") );
    }
    menuattached=SetMenuStrip(window,menu);
    DI( printf("window.c: setupWindow() SetMenuStrip() = %lx\n",menuattached) );
    refreshWindow();
}
EXIT:
}
/*====! Blank.id/shutdownWindow() =====*/
*   NAME
*   shutdownWindow -- Perform the steps necessary to close the window.
*
*   SYNOPSIS
*   shutdownWindow()
*   VOID shutdownWindow(VOID);
*
*   FUNCTION
*   Closes the window and remembers its position for the next open.
*
*   INPUTS
*
*   RESULT
*
*   EXAMPLE
*
*   NOTES
*
*   BUGS
*
*   SEE ALSO
*
* .....
*/
VOID shutdownWindow()
{
    WORD *wp;

    DI( printf("window.c: shutdownWindow() enter\n") );
    if( ! window )
    {
        DI( printf("window.c: shutdownWindow() window not open!\n") );
        return;
    }
    /* save window position */
    *wp++ = saveWindow;
    *wp++ = window->LeftEdge;
    *wp++ = window->TopEdge;
    *wp++ = window->Width;
    *wp = window->Height;
    if(menuattached)
    {
        ClearMenuStrip(window);
        menuattached=NULL;
    }
    if(menu)

```

```

    {
        FreeMenus(menu);
        menu=NULL;
    }

    DI( printf("window.c: shutdownWindow() ClosingWindow(%lx)\n",window) );
    CloseWindow(window);
    DI( printf("window.c: shutdownWindow() after CloseWindow()\n") );
    window = NULL;
    !port = NULL;
    !isqflag = NULL;

    removeGadgets();
    DI( printf("window.c: shutdownWindow() after removeGadgets()\n") );

    {
        if(v1)
        {
            DI( printf("window.c: shutdownWindow() FreeVisualInfo(%lx)\n",v1) );
            FreeVisualInfo(v1);
            v1=NULL;
        }
        if(mydl)
        {
            DI( printf("window.c: shutdownWindow() FreeScreenDrawInfo(%lx)\n",mydl) );
            FreeScreenDrawInfo(myScreen,mydl);
            mydl=NULL;
        }
        if(myScreen)
        {
            DI( printf("window.c: shutdownWindow() UnlockPubScreen(%lx)\n",myScreen) );
            UnlockPubScreen(NULL,myScreen);
            myScreen=NULL;
        }
    }

    /***! Blank.Id/getNewWindow() *****
NAME
    getNewWindow -- Open the window remembering the old position
    If reopening.
SYNOPSIS
    window = getNewWindow();
    struct Window *getNewWindow(VOID);
FUNCTION
    This function opens the commodities window. It automatically
    sets the window title to reflect the current Hotkey. If the
    window has already been opened once then the previous position
    and size (if sizing is enabled) are used for this open.
INPUTS
    None.
RESULT
    Returns a pointer to the opened window or NULL on error.
EXAMPLE
NOTES
BUGS
SEE ALSO
    setupWindow();

```

```

    shutdownWindow();
    *****
    */
    struct Window *getNewWindow()
    {
        struct NewWindow nw;
        WORD *wp = saveWindow;

        DI( printf("window.c: getNewWindow() enter\n") );

        sprintf(WindowTitle,"%s: Hotkey=%s",COM_TITLE,hotkeybuff);

        nw.LeftEdge = *wp++;
        nw.TopEdge = *wp++;
        nw.Width = *wp++;
        nw.Height = *wp++;
        nw.DetailPen = (UBYTE) -1;
        nw.BlockPen = (UBYTE) -1;
        nw.IDCPFlags = WFLAGS;
        nw.Flags = NULL;
        nw.FirstGadget = NULL;
        nw.CheckMark = NULL;
        nw.Title = WindowTitle;
        nw.Screen = NULL;
        nw.BitMap = NULL;
        nw.MinWidth = WINDOW_MIN_WIDTH;
        nw.MinHeight = WINDOW_MIN_HEIGHT;
        /* work around bug */
        nw.MaxWidth = WINDOW_MAX_WIDTH;
        nw.MaxHeight = WINDOW_MAX_HEIGHT;
        nw.Type = WBSCHSCREEN;

        DI( printf("window.c: getNewWindow() before OpenWindowTag()\n") );
        return ((struct Window *) OpenWindowTag(&nw,
            MA_InnerHeight,WINDOW_INNERHEIGHT,
            MA_AutoAdjust,TRUE,MA_PubScreen,myScreen,TAG_DONE));
    }

    /***! Blank.Id/addGadgets() *****
NAME
    addGadgets -- Add all the standard and application specific
    gadgets to the window.
SYNOPSIS
    result = addGadgets(window);
    int addGadgets(struct Window *window);
FUNCTION
    Sets up the environment for gadget toolkit gadgets and calls
    addCustomGadgets() to add the application specific gadgets
    to the window.
INPUTS
    window = Pointer to the window.
RESULT
    Returns TRUE if all went well, FALSE otherwise.
EXAMPLE
NOTES

```

```

* BUGS
*
* SEE ALSO
*   setupCustomGadgets(),
*   removeGadgets(),
*
* .....
*/
int addGadgets(struct Window *window)
{
    struct Gadget *gad;

    DI( printf("window.c: addGadgets() enter\n") );

    /* Open desired font: */
    if( ! font )
    {
        DI( printf("window.c: addGadgets() Opening font\n") );
        if( ! (font = OpenFont(tmydesiredfont)) )
        {
            DI( printf("window.c: addGadgets() Could not open font\n") );
            return(FALSE);
        }
    }

    gad = CreateContext(&glist);
    setupCustomGadgets(&gad);

    if(!gad)
    {
        DI( printf("window.c: addGadgets() error adding gadgets\n") );
        if(glist)
        {
            FreeGadgets(glist);
            glist=NULL;
        }
        if(font)
        {
            CloseFont(font);
            font=NULL;
        }
        return(FALSE);
    }

    AddGlist(window, glist, (UMORD) -1, (UMORD) -1, NULL);
    RefreshGlist(window->firstgadget, window, NULL, ((UMORD) -1));
    GI_RefreshWindow(window, NULL);
    return(TRUE);
}

/***1* Blank.id/removeGadgets() .....
NAME
    removeGadgets -- Remove and deallocate all standard and
                    application gadgets from the window.
SYNOPSIS
    removeGadgets()
    VOID removeGadgets(VOID);
FUNCTION
    This function performs gadget cleanup. It is responsible for
    deallocating and removing all gadgets from the window before
    it is closed.

```

```

* INPUTS
*   None.
*
* RESULT
*   All gadgets are freed and removed from the window.
*
* EXAMPLE
*
* NOTES
*   Uses the global variable glist which is a linked list of all
*   the gadget toolkit gadgets.
*
* BUGS
*
* SEE ALSO
*
* .....
*/
void removeGadgets()
{
    if(glist)
    {
        DI( printf("window.c: removeGadgets() FreeingGadgets(%lx)\n",glist) );
        FreeGadgets(glist);
        glist=NULL;
    }
    if(font)
    {
        DI( printf("window.c: removeGadgets() Closing font %lx\n",font) );
        CloseFont(font);
        font=NULL;
    }
}

#endif /* WINDOW */

```



```

/*****
 *
 *      COPYRIGHTS
 *
 *      UNLESS OTHERWISE NOTED, ALL FILES ARE
 *      Copyright (c) 1990 Commodore-Amiga, Inc. All Rights Reserved.
 *
 *****/

#include <clib/all_protos.h>
#include <pragmas/gadtools_pragmas.h>
#endif

#include <utility/cgitem.h>

/*****
 *
 *      Prototypes for functions declared in app.c and called from the
 *      standard modules.
 *
 *****/
void setupCustomGadgets(struct Gadget **);
void HandleGadget(ULONG,ULONG);
void setupCustomMenu(VOID);
void handleCustomMenu(UMORD);
void refreshWindow(VOID);
bool setupCustomCX(VOID);
void shutdownCustomCX(VOID);
void handleCustomCXMsg(ULONG);
void handleCustomCXCommand(ULONG);
void handleCustomSignal(VOID);

/*****
 *
 *      Prototypes for functions declared in the standard modules and
 *      called by app.c
 *
 *****/
void setupWindow(VOID);
void shutdownWindow(VOID);
void terminate(VOID);

/*****
 *
 *      Prototypes for functions declared in application modules and
 *      called by app.c
 *
 *****/
/*****
 *
 *      definitions for global variables declared in the standard modules
 *      referenced by app.c
 *****/
extern CxObj
extern SHORT
extern VOID
extern struct Menu
extern struct Library
extern struct Gadget
extern char
extern struct MsgPort
extern struct IntuitionBase
extern struct DrawInfo
extern ULONG
extern struct Task
extern bool

    broker;
    topborder;
    *v1;
    *menu;
    *gadtoolBase;
    *glst;
    *types;
    *export;
    *intuitionBase;
    *mydl;
    *cglst;
    *maltask;
    IDCMPRefresh;

```

```

/*****
 *
 *      definitions for global variables declared in app.c and
 *      referenced by the standard modules.
 *
 *****/
extern struct TextAttr mydeslfont;

/*****
 *
 *      Commodities specific definitions.
 *
 *****/
/* COM NAME - used for the scrolling display in the Exchange program */
/* COM_TITLE - used for the window title bar and the long description */
/* COM_DESC - Commodities description used by the Exchange program
   In the Exchange program
/* CX_DEFAULT_PRIORITY - default priority for this commodities broker
   can be overridden by using icon TOOL TYPES
 *****/
#define COM_NAME "Blank"
#define COM_TITLE "Blank"
#define COM_DESCR "Commodities Application Skeleton"
#define CX_DEFAULT_PRIORITY 0
#define CX_DEFAULT_POP_NEX ("shift f1")
#define CX_DEFAULT_POP_ON_START ("YES")

/*****
 *
 *      Custom signal control
 *
 *****/
/* If SIGNAL = 0 then this commodity will NOT have a custom signal
 * If SIGNAL = 1 this commodity will support a custom signal
 *****/
#define SIGNAL 0

/*****
 *
 *      Window control
 *
 *****/
/* If WINDOW = 0 then this commodity will NOT have a popup window
 * If WINDOW = 1 this commodity will support a popup window with the
   attributes defined below.
 *****/
#define WINDOW 1

/* If WINDOW
   define W(s) x
   false
   define W(s) /
   sendif

*****/
/* If WINDOW
extern struct Window *window; /* our window */
extern struct TextFont *font;

#define WINDOW_LEFT 134
#define WINDOW_TOP 64
#define WINDOW_WIDTH 362
#define WINDOW_HEIGHT 68
#define WINDOW_INNERHEIGHT 70

#define WINDOW_SIZING 0
/* If WINDOW_SIZING
#define WINDOW_MAX_WIDTH -1
#define WINDOW_MIN_WIDTH 50
#define WINDOW_MAX_HEIGHT -1
#define WINDOW_MIN_HEIGHT 30
#define WINDOW_FLAGS (ACTIVATE | WINDOWCLOSE | WINDOWDRAG | WINDOWRESIZING | WINDOWDEPTH | SI
MPLE_REFRESH )
false

```

```

#define WINDOW_MAX_WIDTH  WINDOW_WIDTH
#define WINDOW_MIN_WIDTH  WINDOW_WIDTH
#define WINDOW_MAX_HEIGHT WINDOW_HEIGHT
#define WINDOW_MIN_HEIGHT WINDOW_HEIGHT
#define WFLAGS (ACTIVATE | WINDOWCLOSE | WINDOWDRAG | WINDOWDEPTH | SIMPLE_REFRESH )
#endif /* WINDOW_SIZING */

#define IFLAGS (MENUPICK | MOUSEBUTTONS | GADGETUP | GADGETDOWN | MOUSEMOVE | CLOSEMI
NDOW | REFRESHWINDOW )

#define POP_KEY_ID  (66L)      /* hotkey definitions
                               /* pop up identifier */

/* Gadget control
/*
/* Here are the gadget specific definitions. Note that these are
/* included only if WINDOW=1 since gadgets make no sense without a
/* window.
/*
#define GAD_HIDE  1
#define GAD_DIE   2

/* Menu control
/*
/* Here are the menu specific definitions. Note that these are
/* included only if WINDOW=1 since menus make no sense without a
/* window.
#define MENU_HIDE  1
#define MENU_DIE   2

#endif /* WINDOW */

/* Debug control
/*
/* The first define converts any printf that got in by mistake into
/* kprintf. If you are debugging to the console you can change
/* kprintf into printf.
/* The D(x) define controls debugging in the standard modules. Use
/* The D(x) macro for debugging in the app.c and application modules.
/*
void kprintf(char "...");
#define printf kprintf

#ifdef DEBUG
#define D(x) x
#define D(x) x
#else
#define D(x) ;
#define D(x) ;
#endif /* NO_DEBUG */

#endif /* APP_H */

```

```

/*****
 *
 *      COPYRIGHTS
 *
 *      UNLESS OTHERWISE NOTED, ALL FILES ARE
 *      Copyright (c) 1990 Commodore-Amiga, Inc.  All Rights Reserved
 *
 *****/

/* app.c This file contains the custom code for a commodity */
/* you should be able to write a new commodity by changing only */
/* app.c and app.h */

#include "app.h"

#ifdef WINDOW

#define V(x) ((VOID *)x)

struct TextAttr mydesiredfont =
{
    "topaz.font", /* Name */
    8, /* YSize */
    0, /* Style */
    0, /* Flags */
};

VOID setupCustomGadgets(gad)
struct Gadget *gad;
{
    struct MenuGadget mgr;
    ng.ng_VisualInfo=V1;

    ng.ng_TopEdge = topborder+0;
    ng.ng_LeftEdge = 10;
    ng.ng_Width = 40;
    ng.ng_Height = 12;
    ng.ng_GadgetText = "Hide";
    ng.ng_TextAttr = mydesiredfont;
    ng.ng_GadgetID = GAD_HIDE;
    ng.ng_Flags = NULL;
    ng.ng_VisualInfo = V1;
    *gad = CreateGadget(BUTTON_KIND,*gad, mgr, TAG_DONE);

    ng.ng_TopEdge = topborder+15;
    ng.ng_LeftEdge = 10;
    ng.ng_Width = 40;
    ng.ng_Height = 12;
    ng.ng_GadgetText = "Quit";
    ng.ng_TextAttr = mydesiredfont;
    ng.ng_GadgetID = GAD_DIE;
    ng.ng_Flags = NULL;
    ng.ng_VisualInfo = V1;
    *gad = CreateGadget(BUTTON_KIND,*gad, mgr, TAG_DONE);

    VOID HandleGadget(gad,code)
    ULONG gad,code;
    {
        D( kprintf("app: HandleGadget (%lx)\n",gad) );
        switch(gad)
        {
            case GAD_HIDE:
                D( kprintf("app: HandleGadget () GAD_HIDE\n"), );
                shutdownWindow();
        }
    }
}

```

```

        break;
    case GAD_DIE:
        D( kprintf("app: HandleGadget() GAD_DIE\n"), )
        terminate();
    }
}

VOID setupCustomMenu()
{
    struct NewMenu mynewmenu [] =
    {
        { NM_TITLE, "Project", 0, 0, 0, 0, V(MENU_HIDE), },
        { NM_ITEM, "Hide", "H", 0, 0, V(MENU_HIDE), },
        { NM_ITEM, "Quit", "Q", 0, 0, V(MENU_DIE), },
        { NM_END, 0, 0, 0, 0, 0, },
    };
}

menu=CreateMenus(mynewmenu,TAG_DONE);
D( kprintf("app: CreateMenus returns menu = %lx\n",menu); )

VOID handleCustomMenu(code)
WORD code;
{
    struct MenuItem *item;
    BOOL terminated=FALSE;

    D( kprintf("app: handleCustomMenu(code=%lx)\n",code); )
    while((code!=MENU_NULL)&&(terminated))
    {
        item=itemAddress(menu,code);
        switch( (int)MENU_USERDATA(item) )
        {
            case MENU_HIDE:
                shutdownWindow();
                terminated=TRUE; /* since window is gone NextSelect is invalid so... */
                break;
            case MENU_DIE:
                terminate();
                break;
            default:
                break;
        }
    }
    code=item->NextSelect;
    D( kprintf("app: handleCustomMenu next code=%lx\n",code); )
}
D( kprintf("app: handleCustomMenu exits"); )
}
VOID refreshWindow()
{
    if(window)
    {
        if(IDCMPRefresh)
            GI_BeginRefresh( window );

        setApen(window->Report, (UINTZ)mydi->dri Pens(hiliteTextPen));
        setBpen(window->Report, (UINTZ)mydi->dri Pens(hilitePen));
        setDend(window->Report,JMZ);
        setFont(window->Report,font);
        Move(window->Report,90,(WORD)(topborder+40));
        Text(window->Report,"Your Imagery Here",17);

        setApen(window->Report, (UINTZ)mydi->dri Pens(hiliteTextPen));
        setBpen(window->Report, (UINTZ)mydi->dri Pens(backgroundPen));
        Move(window->Report,10,(WORD)(topborder+40));
        Text(window->Report,"HiLight:",8);
    }
}
if(IDCMPRefresh)

```

```
    GT_EndRefresh( window, IL );  
    /* It is possible that the user has selected a font so large */  
    /* that our imagery will fall off the bottom of the window */  
    /* we RefreshWindowFrame here in case our borders were overwritten */  
    if((topborder+WINDOW_INNERHEIGHT) > window->Height)  
        RefreshWindowFrame(window);  
    }  
    return;  
}  
#endif /* WINDOW */  
BOOL setupCustomCX()  
{  
    return(1);  
}  
VOID shutdownCustomCX()  
{  
    VOID handleCustomCXMsg(1d)  
    ULONG 1d;  
    {  
        switch(1d)  
        {  
            case 0:  
                default:  
                    break;  
        }  
    }  
    VOID handleCustomCXCommand(1d)  
    ULONG 1d;  
    {  
        switch(1d)  
        {  
            case 0:  
                default:  
                    break;  
        }  
    }  
    #if CSIGNAL  
    VOID handleCustomSignal(VOID)  
    {  
    }  
    #endif  
}
```

```

#define LIBRARIES_COMMODITIES_H
#define LIBRARIES_COMMODITIES_H

#include <exec/types.h>
#endif

/*****
 * object creation macros
 *****/

#define CxFilterId(d) CreateCxBj((LONG)CX_FILTER, (LONG)d, 0)
#define CxTypeFilter(type) CreateCxBj((LONG)CX_TYPEFILTER, (LONG)type, 0)
#define CxSender(port, id) CreateCxBj((LONG)CX_SEND, (LONG)port, (LONG)id)
#define CxSignal(task, sig) CreateCxBj((LONG)CX_SIGNAL, (LONG)task, (LONG)sig)
#define CxTranslate(id) CreateCxBj((LONG)CX_TRANSLATE, (LONG)id, 0)
#define CxDebug(id) CreateCxBj((LONG)CX_DEBUG, (LONG)id, 0)
#define CxCustom(action, id) CreateCxBj((LONG)CX_CUSTOM, (LONG)action, (LONG)id)

/*****
 * Broker stuff
 *****/

/* buffer sizes */
#define CBD_NAMELEN 24
#define CBD_TITLELEN 40
#define CBD_DESCLEN 40

/* CxBroker errors */
#define CxERR_OK 0 /* No error
#define CxERR_SYERR 1 /* System error, no memory, etc
#define CxERR_DUP 2 /* uniqueness violation
#define CxERR_VERSION 3 /* didn't understand nb_VERSION
#define NB_VERSION 5 /* Version of NewBroker structure

struct NewBroker {
    BYTE nb_Version; /* set to NB_VERSION
    BYTE nb_Name;
    BYTE nb_Title;
    BYTE nb_Descr;
    SHORT nb_Uniques;
    SHORT nb_Flags;
    BYTE nb_Pri;
    /* new in V5
    struct MapPort *nb_Port;
    WORD nb_ReservedChannel; /* plans for later port sharing
};

/* flags for nb_Uniques */
#define NBU_DUPLICATE 0
#define NBU_UNIQUE 1 /* will not allow duplicates
#define NBU_NOTIFY 2 /* sends CXM_UNIQUE to existing broker
/* flags for nb_Flags */
#define CBF_SHOW_HIDE 4

/*****
 * cxusr
 *****/

/* Fake data types for system private objects */
typedef CX_H CxObj;
typedef LONG CxMsg;
#endif

```

```

/* Pointer to function returning Long */
typedef LONG (*PFL)();

/*****
 * Commodities Object Types
 *****/

#define CX_INVALID 0 /* not a valid object (probably null)
#define CX_FILTER 1 /* input event messages only
#define CX_TYPEFILTER 2 /* filter on message type
#define CX_SEND 3 /* sends a message
#define CX_SIGNAL 4 /* sends a signal
#define CX_TRANSLATE 5 /* translates IE into chain
#define CX_BROKER 6 /* application representative
#define CX_DEBUG 7 /* dumps kprintf to serial port
#define CX_CUSTOM 8 /* application provides function
#define CX_ZERO 9 /* system terminator node

/*****
 * CxMsg types
 *****/

#define CXM_UNIQUE (1 << 4) /* sent down broker by CxBroker()
/* obsolete: subsumed by CXM_COMMAND (below)
/* Messages of this type rattle around the Commodities input network.
* They will be sent to you by a sender object, and passed to you
* as a synchronous function call by a Custom object.
* The message port or function entry point is stored in the object,
* and the ID field of the message will be set to what you arrange
* issuing object.
* The Data field will point to the input event triggering the
* message.
#define CXM_EVENT (1 << 5)

/* These messages are sent to a port attached to your Broker.
* They are sent to you when the controller program wants your
* program to do something. The ID field identifies the command.
* The Data field will be used later.
#define CXM_COMMAND (1 << 6)

/* ID values
#define CXCMD_DISABLE (15) /* please disable yourself
#define CXCMD_ENABLE (17) /* please enable yourself
#define CXCMD_APPEAR (19) /* open your window, if you can
#define CXCMD_DISAPPEAR (21) /* go dormant
#define CXCMD_KILL (23) /* go away for good
#define CXCMD_UNIQUE (25) /* someone tried to create a broker
* with your name. Suggest you Appear.

#define CXCMD_LIST_CHG (27) /* Used by Exchange program. Someone
/* has changed the broker list

/* return values for BrokerCommand():
#define CMD_OK (0)
#define CMD_NOBROKER (-1)
#define CMD_NOPORT (-2)
#define CMD_NOMEM (-3)

/* IMPORTANT NOTE: for V5:
* Only CXM_EVENT messages are passed through the input network.
* Other types of messages are sent to an optional port in your broker.

```

```

* This means that you must test the message type in your message handling,
* If input messages and command messages come to the same port.
* Older programs have no broker port, so processing loops which
* make assumptions about type won't encounter the new message types.
* The TypeFilter CxObject is hereby obsolete.
* It is less convenient for the application, but eliminates testing
* for type of input messages.
*/

/*****
** CxObj Error flags (return values from CxObjError()) **/
/*****
#define CORR_ISNULL 1 /* you called CxError(NULL) */
#define CORR_NULLATTACH 2 /* someone attached NULL to my list */
#define CORR_BADFILTER 4 /* a bad filter description was given */
#define CORR_BADTYPE 8 /* unmatched type-specific operation */

/****
* ix
****/

#define IX_VERSION 2

struct InputXpression {
    UBYTE ix_Version; /* must be set to IX_VERSION */
    UBYTE ix_Class; /* class must match exactly */
    UWORD ix_Code; /* normally used for UPCODE */
    UWORD ix_CodeMask; /* normally used for UPCODE */
    UWORD ix_Qualifier; /* synonyms in qualifier */
    UWORD ix_QualifierMask; /* synonyms in qualifier */
};

typedef struct InputXpression IX;

/* QualSame Identifiers */
#define IXSYM_SHIFT 1 /* left- and right- shift are equivalent */
#define IXSYM_CAPS 2 /* either shift or caps lock are equivalent */
#define IXSYM_ALT 4 /* left- and right- alt are equivalent */

/* corresponding QualSame masks */
#define IXSYM_SHIFTMASK (IEQUALIFIER_SHIFT | IEQUALIFIER_RSHIFT)
#define IXSYM_CAPSMASK (IXSYM_SHIFTMASK | IEQUALIFIER_CAPSLOCK)
#define IXSYM_ALTMASK (IEQUALIFIER_ALT | IEQUALIFIER_RALT)

#define IX_NORMALQUALS 0x7fff /* for QualMask field: avoid RELATIVEHOUSE */

/* matches nothing */
#define NULL_IX(1) ((1) -> ix_Class == IECLASS_NULL)

#endif

```




Internationalization of Software: the locale.library

by Valentin Pepelea

*Preliminary: the information in this article is subject to change without notice.
The locale.library is planned for a future version of the Amiga OS.*

The *locale.library* will be the cornerstone of an internationalized Amiga operating system Commodore is planning. The library presented here is not in its final form, and your suggestions on how to resolve open issues are welcome.

Introduction

Presently, developers must write a special version of their software package for each country in which they intend to market it. This task could be difficult depending on how many details have to be taken into account. It is often not sufficient to translate the strings into the target language – many other items have to be taken into account:

- ☐ the time and date formats used in the country
- ☐ the money symbols and denomination
- ☐ the method of grouping numbers
- ☐ the order in which strings must be sorted (collation)
- ☐ the order in which components must appear within a string.

Objective

Several other internationalization standards have appeared in recent years. Our goal is not to implement any particular one, but rather to draw elements from each of them as they best fit our operating system. The Amiga is blessed with a tight and efficient kernel, and we strive to keep it that way.

While not implementing any existing standard, we also try not to prevent third-party implementations. Of particular interest to some may be the ANSI C and X/Open NLS standards, as implemented in SVR4 for example. The *locale.library* contains enough information to allow compiler writers to fully implement the functions in those standards.

Categories

The `locale.library` is composed of data structures that define the cultural environment of the user, as well as a limited set of functions that manipulate these structures or behave according to the information contained within.

An individual's cultural locale can be divided into two categories:

- ☐ The territory category comprises the time, date, currency and numerical format definitions which depend on which country, state or territory the user lives in.
- ☐ The language category defines the character set, collation information and strings which vary according to the language or dialect of the user.

The `locale.library` contains functions, `OpenLanguage()` and `OpenTerritory()`, which load in memory and return a pointer to corresponding language and territory structures. These pointers may be used as parameters for other locale functions; or the information contained within the structures may be used directly by the programmer.

By not defining a global default language and territory, the programmer may write applications that allow the user to independently select the elements of his locale. For example, an Swiss civil engineer educated in England could use a planning package that allows him to write comments in German, draw plans and list measurement in meters, calculate the costs in U.S. dollars, and print the time and dates of milestones in typical Swiss fashion. Meanwhile, he may prefer dealing with operating system commands and messages in English.

Thus while this enginner might have used a preferences editor to set his operating system default to the English language and Swiss territory, the application he used allowed him to ignore these selections and provided him with the option of using something different for:

- ☐ measurement
- ☐ language
- ☐ currency
- ☐ time and dates.

The `CloseTerritory()` and `CloseLangauge()` functions inform the system that access to these structures have completed, and they may be flushed out of memory.

Commodore will be providing language and territory tables for the countries in which it sells computers. Programs that generate such tables will be provided for those wishing to create their own.

Message catalogues

In order to facilitate the translation of their software packages, programmers may put all their strings in a special file, and retrieve them by number. This way even users may translate the software they own by simply modifying these message catalogues. The `CatOpen()` function loads a message catalogue in memory, and pointers to specific strings can be obtained using the `CatGets()` function. These are defined in the X/Open standard.

The `CatClose()` function informs the system that access to the specified catalogue has concluded and it may be flushed out if its open count reaches zero. There are cases though, when even if the open count reaches zero, we may want to keep the catalogue in memory. The commands in the `c:` directory, for example, would be less responsive if they had to load their catalogues every time they were launched. For such cases the `CAT_PERMANENT` flag may be set, preventing flushing when the open count reaches zero.

Preferences

A Preferences editor will be provided to allow users to select their preferred territory (country) and language. Workbench disks will come pre-set with the territory and language of the country in which they are sold. These selections are expected to be static. Operating system commands and applications that have already started are not expected to change their behavior if the user changes his preferred territory or language meanwhile.

The *locale*: directory contains all files pertaining to localization. The files contained in *locale*: are territory tables while the directories are language directories in which message catalogues can be found. One such catalogue named "date-ctype-collation" contains the strings used in the long date format, information on the character set used by the language, and the collation information on the language's alphabet.

Third-party applications may also put their catalogues in the *locale:<language>* directories, but it is recommended to leave such catalogues in the application directory itself. It is simpler to find out which catalogues are available that way.

The functions `DefaultLanguage()` and `DefaultTerritory()` will return pointers to the DOS defaults, and the literal name of the language or territory can be found out by the `Node.In_name` field of these structures. The operating system will use the pointers returned by these functions as parameters to display information according to the user's locale.

(Open issue: should the files and directories' names be in English or in their own language? What if the language, such as Chinese, uses a different character set and font?)

Collation Information

Two functions are provided in the locale library which make use of collation information from a language table: the ANSI C `StrColl()` and the `StrXCmp()` functions.

`StrColl()` compares two strings according to the rules of the language in use. `StrXFrn()` transforms a string in such a way that if two strings are processed through this function, they may be compared using the old `strcmp()` function and case the same result as if `StrColl()` had been applied to the original strings.

The `*collation` fields in the language table either point to a three-pass collation table, or to the `StrColl()` and `StrXFrn()` functions themselves. Some languages might have collation rules that cannot be implemented correctly by using tables – instead they need some rules to be hardcoded in the comparison functions.

Argument Ordering

The grammar of most languages requires that subjects, objects, adverbs and complements be written out in a specific order. Unfortunately, the order differs from language to language. For example, “Put volume ‘Hello’ in drive DF0:” might be said as “Put in drive DF0:, volume ‘Hello’” in another language.

The `Exec RawDoFmt()` function is therefore updated to support a new formatting argument, “X\$”, where X specifies the order of the argument. Here is an example usage:

```
printf("Put %d dollars in box number %d.",10,3);
Put 10 dollars in box lumber 3.

printf("In box number %2$d, put %1$ dollars.",10,3);
In box number 3, put 10 dollars.
```

Detailed Description of Structures

Monetary and numeric information. The information is stored in the `lconv` structure, as defined by ANSI C in `<locale.h>`. Although it is possible to pack this information in a more compact format, compiler writers would then have to write a `localeconv()` function which takes our format and transforms it into the struct `lconv` format. Storing the information in this larger, but complete, manner from the start is preferable. Here are the elements of `lconv`:

<code>char *decimal_point</code>	The decimal point character used to format non-monetary quantities.
----------------------------------	---

<code>char *thousands_sep</code>	The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.
<code>char *grouping</code>	A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.
<code>char *int_curr_symbol</code>	The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217 Codes for the Representation of Currency and Funds. The fourth character (immediately preceding the NULL) is the character used to separate the international currency symbol from the monetary quantity.
<code>char *currency_symbol</code>	The currency symbol applicable to the current locale.
<code>char *mon_decimal_point</code>	The decimal point used to format monetary quantities.
<code>char *mon_thousands_sep</code>	The separator for groups of digits before the decimal point in formatted monetary quantities.
<code>char *mon_grouping</code>	A string whose elements indicate the size of each group of digits in formatted monetary quantities.
<code>char *positive_sign</code>	The string used to indicate a nonnegative-valued formatted monetary quantity.
<code>char *negative_sign</code>	The string used to indicate a negative-valued formatted monetary quantity.
<code>char int_frac_digits</code>	The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.
<code>char frac_digits</code>	The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.
<code>char p_cs_precedes</code>	Set to 1 or 0 if the <code>currency_symbol</code> respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.
<code>char n_cs_precedes</code>	Set to 1 or 0 if the <code>currency_symbol</code> respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
<code>char p_sign_posn</code>	Set to a value indicating the positioning of the <code>positive_sign</code> for a non-negative formatted monetary quantity.
<code>char n_sign_posn</code>	Set to a value indicating the positioning of the <code>negative_sign</code> for a negative formatted monetary quantity.

The elements `*grouping` and `*mon_grouping` are interpreted according to the following:

<code>CHAR_MAX</code>	No further grouping is to be performed.
<code>0</code>	The previous element is to be repeatedly used for the remainder of the digits.
<code>other</code>	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

<code>0</code>	Parentheses surround the quantity and <code>currency_symbol</code> .
<code>1</code>	The sign string precedes the quantity and <code>currency_symbol</code> .
<code>2</code>	The sign string succeeds the quantity and <code>currency_symbol</code> .
<code>3</code>	The sign string immediately precedes the <code>currency_symbol</code> .
<code>4</code>	The sign string immediately succeeds the <code>currency_symbol</code> .

In addition to the information contained in the `lconv` structure, the Amiga locale table includes two additional entries:

<code>char *small_currency_symbol</code>	The small currency symbol applicable to the current locale.
<code>char metric</code>	The measuring system applicable in this locale. <code>0</code> The international metric system <code>1</code> The U.S. system <code>2</code> The Imperial system <code>4</code> The British system

Time and Date Formats

The way in which the time and date formats are specified aims to be easily expandable. The following formats are already supported, and as others are discovered, they can be added:

Date	Time
1990-07-04	3:30PM
4.7.90	1530
7/4/76	15h.30
4.VII.76	15.30
90186	15:30

<code>char short_date_1</code>	Identifies the first element of the short date format.
<code>char short_date_2</code>	Identifies the second element of the short date format.
<code>char short_date_3</code>	Identifies the third element of the short date format.

<code>char *date_sep_1</code>	The symbol used to separate the first and second elements of the short date string.
<code>char *date_sep_2</code>	The symbol used to separate the second and third elements of the short date string.
<code>char long_date_1</code>	Identifies the first element of the long date format.
<code>char long_date_2</code>	Identifies the second element of the long date format.
<code>char long_date_3</code>	Identifies the third element of the long date format.
<code>char long_date_4</code>	Identifies the fourth element of the long date format.
<code>char *long_date_sep_1</code>	The symbol used to separate the first and second elements of the long date string.
<code>char *long_date_sep_2</code>	The symbol used to separate the second and third elements of the long date string.
<code>char *long_date_sep_3</code>	The symbol used to separate the third and fourth elements of the long date string.
<code>char time_1</code>	Identifies the first element of the time format.
<code>char time_2</code>	Identifies the second element of the time format.
<code>char time_3</code>	Identifies the third element of the time format.
<code>char *time_separator_1</code>	The symbol used to separate the first and second elements of the time string.
<code>char *time_separator_2</code>	The symbol used to separate the second and third elements of the time string.

The elements `short_date` and `long_date` are interpreted according to the following:

- 0 This element is to be left blank.
- 1 Full name of day.
- 2 Abbreviated name of day.
- 3 Day of month, with leading zeros. (04 Jan) [1-31]
- 4 Day of month, without leading zeros. (4 Jan)
- 5 Day of year, with leading zeros. [1-366]
- 6 Day of year, without leading zeros.
- 7 Day of week. [1-7]
- 8 Full name of month.
- 9 Abbreviated name of month.
- 10 Month of the year, with leading zeros. (4-04-90)
- 11 Month of the year, without leading zeros. (4-4-90)
- 12 Month of the year, in Roman numeral format.
- 13 Year number, without century. (90) [0-99]
- 14 Year number, with century. (1990) [0-65536]
- 15 Week number of the year, with leading zeros. [0-53]

The elements of the time format identifiers are interpreted as follows:

- 0 This element of to be left blank.
- 1 Hour of the day, in 24 hour format, with leading zeros. (03:30)
- 2 Hour of the day, in 24 hour format, without leading zeros.
- 3 Hour of the day, in 12 hour format, with leading zeros.
- 4 Hour of the day, in 12 hour format, without leading zeros.
- 5 Minute of the hour, with leading zeros.
- 6 Minute of the hour, without leading zeros.
- 7 Second of the minute, with leading zeros. [0-61]
- 8 Second of the minute, without leading zeros.

More element formats may be defined as the need arises.

Character Type Information

Through the use of the keymap, the user may assign any character to any keyboard key and thus be able to enter any character he wishes. Still there is need for character type information for such functions as `isalpha()`, `toupper()`, `tolower()`, etc.

There is currently no explicit support for multi-byte character sets nor wide characters, but this implementation does not preclude such support. Bit 0 of the `Flags` entry in the language structure indicates whether an 8-bit character set or some other mechanism is being used.

The Amiga currently assumes that the ECMA-94 Latin 1 character set is being used; the "/" and ":" characters are hard coded into OS commands and applications. Developers are free to use other character sets for non-filesystem endeavors, but they then have to supply the appropriate character type information table, which is stored as follows:

```
char *is[256]    A character type information table. Information about each
                  character is stored in the corresponding array cell.

char *lower
char *upper      Two lists describing how transformation to upper-case or to
                  lower-case is to be performed.

char space       The code of the space character, (" ").
```

The information in the character type table is stored as follows:

```
isupper = (1<<0)  upper-case character.
islower = (1<<1)  lower-case character.
isdigit = (1<<2)  decimal digit (numeric) character.
isspace = (1<<3)  white space character.
ispunct = (1<<4)  punctuation character.
```

```

iscntrl = (1<<5)    control character.
isprint  = (1<<6)    blank.
isxdigit = (1<<7)    hexadecimal digit.

isalpha  = (1<<0) or (1<<1)
isalnum  = (1<<0) or (1<<1) or (1<<2)
isgraph  = (1<<0) or (1<<1) or (1<<2) or (1<<4)
isprint  = (1<<0) or (1<<1) or (1<<2) or (1<<4) or (1<<6)

```

The `*lower` field is a list of characters, corresponding on a one-to-one basis to the `*upper` list of characters. The lists are started with the lowest valued character and ended with the highest valued character in the list. Individual character correspondence is listed by entering a higher valued character followed by a lower valued character.

For example, in English, the lower-case characters a-z are transformed to A-Z in upper-case. Their lists would then be:

```

*lower:  0x61-0x7A
*upper:  0x41-0x5A

```

In French, there is also the `é` character (`é` accent aigu, 0xC9) that gets transformed to `É` (0xE9). Since this character is not part of a list, it must precede a lower-ordered character:

```

*lower:  0xC9, 0x61 - 0x7A
*upper:  0xE9, 0x41 - 0x5A

```

The following is the list corresponding to the Amiga character set. It should not be used as is with all languages that use this character set because transformation in some languages are different. For example, in French the `é` character might get transformed to `E`, not `É`.

```

*lower:  0xD8 - 0xDE, 0xC0 - 0xD6, 0x61 - 0x7A
*upper:  0xF8 - 0xFE, 0xE0 - 0xF6, 0x41 - 0x5A

```

The `tolower()` and `toupper()` functions are supposed to translate only characters which have their `isupper` and `islower` bits set respectively. The transformation table could thus be simplified to:

```

*lower:  0xC0 - 0xDE, 0x61 - 0x7A
*upper:  0xE0 - 0xFE, 0x41 - 0x5A

```

locale.library AutoDocs

NAME

OpenTerritory -- open a territory table

SYNOPSIS

```
territory = OpenTerritory(name)
    D0                      A0
```

FUNCTION

This function returns a pointer to a territory table that was previously loaded in memory. If the territory table is not already there it will be loaded in from disk. First the directory from which the process was launched is searched, then the locale: directory.

INPUTS

name a pointer to a null terminated string

RESULTS

territory a territory table pointer

SEE ALSO

CloseTerritory, locale/locale.h

/*****

NAME

OpenLanguage -- open a language table

SYNOPSIS

```
language = OpenLanguage(name)
    D0                      A0
```

FUNCTION

This function returns a pointer to a language table, if that table has already been loaded in memory. If not, the directory from which the calling process was launched is searched, then the locale: directory.

INPUT

name a pointer to a null terminated string

RESULTS

languagea language table pointer

SEE ALSO

CloseLanguage, locale/locale.h

/*****

NAME

CloseTerritory -- close a territory table

SYNOPSIS

```
CloseTerritory(territory)
               AO
void CloseTerritory(struct territory *)
```

FUNCTION

This function informs the system that access to the given territory table has concluded. If the open count of the territory table reaches zero, it is removed from memory unless its PERMANENT flag is set.

INPUTS

territory a territory table pointer

SEE ALSO

OpenTerritory, locale/locale.h

/*****

NAME

CloseLanguage -- close a language table

SYNOPSIS

```
CloseLanguage(language)
               AO
void CloseLanguage(struct language *)
```

FUNCTION

This function informs the system that access to the given language table has concluded. If the open count of the language table reaches zero, it is removed from memory unless its PERMANENT flag is set.

INPUTS

language a language table pointer

SEE ALSO

OpenLanguage, locale/locale.h

/*****

NAME

DefaultTerritory -- return the default territory for system func.

SYNOPSIS

```
territory = DefaultTerritory()
struct Territory DefaultTerritory(void)
```

FUNCTION

Returns the default territory used in DOS functions such as DateToStr() and StrToDate(). The name of the territory file can be determined from the territory->Node.ln_Name field.

RESULTS

territory a territory table pointer

SEE ALSO

locale/locale.h

/*****

NAME

DefaultLanguage -- return the default territory for system functions

SYNOPSIS

language = DefaultLanguage()

struct Language DefaultLanguage(void)

FUNCTION

Returns the default language used in DOS functions such as DateToStr() and StrToDate(). The name of the language directory, and thus the language itself, can be determined from the language->Node.ln_Name field.

RESULTS

language a language table pointer

SEE ALSO

locale/locale.h

/*****

NAME

CatOpen -- open a message catalogue

SYNOPSIS

catalogue = CatOpen(name)
D0 A0

FUNCTION

This function returns a pointer to the named catalogue. If it is not already in memory it will first search the directory from which the translation was loaded, and finally the locale: directory.

If the name string contains a ":" character, the locale directory is not searched anymore.

INPUTS

name a pointer to a null terminated string specifying a catalogue.

RESULTS

catalogue a pointer to a message catalogue

SEE ALSO

CatClose(), CatGets(), locale/locale.h

/*****

NAME

CatGets -- get a message from a message catalogue

SYNOPSIS

```
message = CatGets(catalogue, set, msgNum, defaultString)
      DO      A0      DO D1      A1
char *TranslateMsg(struct Catalogue *, int, int, char *);
```

FUNCTION

This function returns a pointer to the message corresponding to the given table and message number. Should the table be a NULL, this function will return the default string given.

INPUTS

table	a translation table
set	a set number, not implmented. Use 1 as set number.
msgNum	a message number

RESULTS

message	a pointer to a null terminated string
---------	---------------------------------------

SEE ALSO

CatOpen(), CatClose(), locale/locale.h

/*****

NAME

CatClose -- close a message catalogue

SYNOPSIS

```
CatClose(catalogue)
      A0
void CatClose(struct Translation *);
```

FUNCTION

This function informs the localisation library that access to the given catalogue has been concluded. If the open count of the catalogue reaches zero, it is removed from memory, unless its PERMANENT flag is set.

INPUTS

catalogue	a pointer to a translation table
-----------	----------------------------------

SEE ALSO

CatOpen(), CatGets(), locale/locale.h

/*****

NAME

StrColl -- compare two strings according to collation information

SYNOPSIS

```
result = StrCmp(string1,string2,language)
D0      A0      A1      A2
int StrCmp(char *, char *, struct Language *)
```

FUNCTION

Compares string1 to string2 according to the collation information provided in the language table and returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by string1 is greater than, equal to, or less than the string pointed to by string2.

INPUTS

string1 a pointer to a null terminated string
string2 a pointer to a null terminated string
language a pointer to a language table

RESULTS

result relationship between string1 and string2

SEE ALSO

locale/locale.h

/*****

NAME

StrXfrm -- transform string according to collation information

SYNOPSIS

```
length = StrXfrm(string1,string2,size,language)
D0      A0      A1      D0      A2
```

FUNCTION

This function transforms the string pointed to by string2 and places the resulting string into the array pointed to by string1. The transformation is such that if the strcmp function is applied to two transformed strings, it returns a value corresponding to the result returned by the StrColl() function applied to the two original strings. No more than 'size' characters are placed into the array pointed to by string1, including the terminating NULL character. If 'size' is zero, string1 is permitted to be a NULL pointer.

INPUTS

string1 a pointer to a null terminated string
string2 a pointer to a null terminated string
size maximum number of characters to be put in string1
language a pointer to a language table

RESULTS

length length of the transformed string. If 'length' is greater than 'size', then string1[] is undetermined.

SEE ALSO

StrColl(), locale/locale.h

/*****

NAME

RawDoFmt -- format data into a character stream.

SYNOPSIS

```
RawDoFmt(FormatString, DataStream, PutChProc, PutChData);
          a0             a1             a2             a3
```

```
void RawDoFmt(STRPTR, APTR, void (*)(), APTR);
```

FUNCTION

perform "C"-language-like formatting of a data stream, outputting the result a character at a time. Where % formatting commands are found in the FormatString, they will be replaced with the corresponding element in the DataStream. %% must be used in the string if a % is desired in the output.

Under V36, RawDoFmt() returns a pointer to the end of the DataStream (The next argument that would have been processed). This allows multiple formatting passes to be made using the same data.

INPUTS

FormatString - a "C"-language-like NULL terminated format string, with the following supported % options:

%[order\$][flags][width.limit][length]type

order - argument number to use for this entry, followed by a "\$".
flags - only one allowed. '-' specifies left justification.
width - field width. If the first character is a '0', the field will be padded with leading 0's.
. - must follow the field width, if specified
limit - maximum number of chars to output from a string (only valid for %s).
length - size of input data defaults to WORD for types d, x and c, 'l' changes this to long (32-bit).
type - supported types are:
b - BSTR, data is 32-bit BPTR to byte count followed by a byte string, or NULL terminated byte string. A NULL BPTR is treated as an empty string. (Added in V36 exec)
d - decimal
x - hexadecimal
s - string, a 32-bit pointer to a NULL terminated byte string. In V36, a NULL pointer is treated as an empty string
c - character

DataStream - a stream of data that is interpreted according to the format string. Often this is a pointer into the task's stack.

PutChProc - the procedure to call with each character to be output, called as:
PutChProc(Char, PutChData);
D0-0:8 A3

The procedure is called with a NULL Char at the end of the format string.

PutChData - a value that is passed through to the PutChProc procedure. This is untouched by RawDoFmt, and may be modified by the PutChProc.

EXAMPLE

```
;
; Simple version of the C "sprintf" function. Assumes C-style
; stack-based function conventions.
;
; long eyecount;
; eyecount=2;
; sprintf(string,"%s have %ld eyes. ","Fish",eyecount);
;
; would produce "Fish have 2 eyes." in the string buffer.
;
XDEF _sprintf
XREF _AbsExecBase
XREF _LVORawDoFmt
_sprintf: ; ( ostring, format, (values) )
    movem.l a2/a3/a6,-(sp)

    move.l 4*4(sp),a3      ;Get the output string pointer
    move.l 5*4(sp),a0      ;Get the FormatString pointer
    lea.l 6*4(sp),a1      ;Get the pointer to the DataStream
    lea.l stuffChar(pc),a2
    move.l _AbsExecBase,a6
    jsr _LVORawDoFmt(a6)

    movem.l (sp)+,a2/a3/a6
    rts

;----- PutChProc function used by RawDoFmt -----
stuffChar:
    move.b d0,(a3)+      ;Put data to output string
    rts
```

WARNING

This Amiga ROM function formats word values in the data stream. If your compiler defaults to longs, you must add an "l" to your % specifications. This can get strange for characters, which might look like "%lc".

SEE ALSO

Documentation on the C language "printf" call in any C language reference book.

dos.library Autodocs (international functions)

```
* NAME
* DateToStr -- Converts a DateStamp to a string
*
* SYNOPSIS
* error = DateToStr(country,datetime)
* DO      AO      D1
* BOOL DateToStr(struct DateTime *)
*
* FUNCTION
* StampToStr converts an AmigaDOS DateStamp to a human readable ASCII string as
* requested by your settings in the DateTime structure.
*
* INPUTS
* country - a pointer to a country table; used only with FORMAT_LOC.
*
* DateTime - a pointer to an initialized DateTime structure. The DateTime
* structure should be initialized as follows:
*
* dat_Stamp - a copy of the datestamp you wish to convert to ascii
*
* dat_Format - a format byte which specifies the format of the dat_StrDate. This
* can be any of the following (note: If value used is something other than
* those below, the default of FORMAT_DOS is used):
*
*     FORMAT_DOS: AmigaDOS format (dd-mmm-yy).
*     FORMAT_INT: Internationalformat (yy-mmm-dd).
*     FORMAT_USA: American format (mm-dd-yy).
*     FORMAT_CDN: Canadian format (dd-mm-yy).
*     FORMAT_DEF: Default format. Used if the localisation library has
*                 not been opened yet FORMAT_DOS is used. Otherwise
*                 the default country table is used.
*     FORMAT_LOC: The country table specified is used for determining the
*                 format, otherwise the country input is ignored.
*
* dat_Flags - a flags byte. The only flag which affects this function is:
*
*     DTB_SUBST: If set, a string such as Today, Monday, etc., will be used
*                 instead of the dat_Format specification if possible.
*     DTB_FUTURE: Ignored by this function.
*
* dat_StrDay - pointer to a buffer to receive the day of the week string.(Monday,
* Tuesday, etc.). If null, this string will not be generated.
*
* dat_StrDate -pointerto a buffer to receive the date string, in the format
* requested by dat_Format, subject to possible modifications by DTB_SUBST.
* If null, this string will not be generated.
*
* dat_StrTime -pointerto a buffer to receive the timeof day string. If NULL,
* this will not be generated.
*
* RESULT
* error - a non-zero return indicates that the DateStamp was invalid, and could
* not be converted. Zero indicates that everything went according to plan.
*
* SEE ALSO
* StrToDate(), libraries/datetime.h
```

```

*   NAME
*   StrToDate -- Converts a string to a DateStamp
*
*   SYNOPSIS
*   error = StrToDate(country,datetime)
*   DO      A0      D1
*   BOOL StrToDate( struct DateTime * )
*
*   FUNCTION
*   Converts a human readable ASCII string into an AmigaDOS DateStamp.
*
*   INPUTS
*   DateTime - a pointer to an initialized DateTime structure.
*   The DateTime structure should be initialized as follows:
*
*   dat_Stamp - ignored on input.
*
*   dat_Format - a format byte which specifies the format of the dat_StrDate. This
*               can be any of the following (note: If value used is something
*               other than those below, the default of FORMAT_DOS is used):
*
*   FORMAT_DOS: AmigaDOS format (dd-mmm-yy).
*   FORMAT_INT: International format (yy-mmm-dd).
*   FORMAT_USA: American format (mm-dd-yy).
*   FORMAT_CDN: Canadian format (dd-mm-yy).
*   FORMAT_DEF: The default format is used. If the localisation library has
*               not been opened yet FORMAT_DOS is used. Otherwise
*               the default country table is used.
*   FORMAT_LOC: The country table specified is used for determining the format,
*               otherwise the country input is ignored.
*
*   dat_Flags - a flags byte. The only flag which affects this function is:
*
*   DTB_SUBST: ignored by this function
*   DTB_FUTURE: If set, indicates that strings such as (stored in dat_StrDate)
*               "Monday" refer to "next" monday. Otherwise, if clear,
*               strings like "Monday" refer to "last" Monday.
*
*   dat_StrDay - ignored by this function.
*
*   dat_StrDate - pointer to valid string representing the date. This can be
*               a "DTB_SUBST" style string such as "Today" "Tomorrow"
*               "Monday", or it may be a string as specified by the dat_Format
*               byte. This will be converted to the ds_Days portion of the
*               DateStamp. If this pointer is NULL, DateStamp->ds_Days
*               will not be affected.
*   dat_StrTime - pointer to a buffer which contains the time in the ASCII format
*               hh:mm:ss. This will be converted to the ds_Minutes and ds_Ticks
*               portions of the DateStamp. If this pointer is NULL, ds_Minutes
*               and ds_Ticks will be unchanged.
*
*   RESULT
*   error - a non-zero return indicates that a conversion could not be performed.
*           A Zero return indicates that the DateTime.dat_Stamp variable contains
*           the converted values.
*
*   SEE ALSO
*   DateToStr(), libraries/datetime.h

```




1 Corrections to Arexx Developer's Conference Notes

During the course of several late night writing sessions, some errors crept into the manuscript for the AREXX developer's Conference Notes. Please excuse the lapse. This correction sheet should clear up the mistakes.

The most glaring error that occurred is in the method for returning results to the program. While it is true that the `rm_Result1` field contains the severity level of the error, the standard use of the `rm_Result2` is that it returns a pointer to a result string. These listings supercede the ones in the notes:

2 Further information on AREXX

There are many functions that will make your life easier. These functions, contained in the AREXX Systems Library, `rexsyslib.library`, which should be in the `libs:` directory.

This means that you need to open it via an `OpenLibrary()` call. The various routines are listed on page 111 of the Arexx User's Reference Manual.


```

-----
doarexx(commstring)
char *commstring;
{
struct RxxMsg *TheRxxMsg, *CreateRxxMsg();

if (TheRxxMessage = CreateRxxMsg(MyPort, "MyExtension", "COMMAND"))
{
    TheRxxMsg->rm_Args[0] = commstring;
    FillRxxMsg(TheRxxMsg, 1, 0x0);
    TheRxxMsg->rm_Action = RXCOMM;

    /* If you want to, initialize rm_StdIn and rm_StdOut to
       redirect these streams for the invoked command
       You may HAVE to do this this routine is in a program that
       was launched from Workbench
    */

    Forbid();
    if(RxxPort = FindPort("REXX"))
        PutMsg(RxxPort, (struct Message *)TheRxxMsg);
    Permit();

    if(RxxPort == NULL)
        puts("Could not open the AREXX Port...");
    else
        return(await(MyPort, TheRxxMsg) ); /* Await also cleans up
                                           TheRxxMsg
                                           */
}
}

```



```

/*
Assume that the message is receives in an outer event processing
loop, and that we merely need to parse it in this routine.
*/

```

```

void
ParseArexxMsg(struct REXXMsg *TheMessage)
{
char *CommandString;
int ArgCounter;
int StringSize;
if(TheMessage->rm_Action != RXCOMM)
{
    TheMessage->rm_Result1 = 10L; /* Serious Error */

    TheMessage->rm_Result2 = 0L; /* if Result1 > 0 then
                                this must be 0
                                */

    return;
}
else
{
    TheMessage->rm_Result1 = Dispatch(TheMessage->rm_Args[0]);
    if(TheMessage->rm_Result1 == 0L)
        TheMessage->rm_Result2 =
CreateArgString(TheReturnString, strlen(TheReturnString));
    else
        TheMessage->rm_Result2 = 0L;

    return;

    /*It is assumed that the message handler front end will
    ReplyMsg() for me.
    */
}
}

```

```

-----

dosshell(commstring)
char *commstring;
{
struct REXXMsg *TheRexxMsg, *CreateRexxMsg();

if (TheRexxMessage = CreateRexxMsg(MyPort, NULL, "COMMAND"))
{
    TheRexxMsg->rm_Args[0] = commstring;
    FillRexxMsg(TheRexxMsg, 1, 0x0);
    TheRexxMsg->rm_Action = RXCOMM | RXFF_STRING;

    Forbid();
    if(RexxPort = FindPort("REXX"))
        PutMsg(RexxPort, (struct Message *)TheRexxMsg);
    Permit();

    if(RexxPort == NULL)
        puts("Could not open the AREXX Port...");
    else
        return(await(MyPort, TheRexxMsg) ); /* Await also cleans up
                                           TheRexxMsg
                                           */
}
}

```

(

(

(

Implementing ARexx In Your Programs

By Kevin Klop

By now, almost all developers of Amiga software must be aware that ARexx is one of the most useful tools for the so-called "power user". This power user will use ARexx for everything from straight scripting in a terminal program to linking two pieces of software together to form a system tailored to his or her criteria.

As an example, these notes were printed using AmigaTeX¹. However, AmigaTeX is by no means a text editor. My favourite text editor can cause ARexx programs to start with the press of a key. AmigaTeX can also understand ARexx. As a result, I bound a simple ARexx program to Right-Amiga-X. This program would save my current file and start AmigaTeX working on formatting it.

Listing 1

ARexx program used constantly to proofread my DevCon Notes

```
/*
 * Example QED <-> TeX ARexx script
 *
 * Does the following:
 *
 * 1.) Save (named) file as 'filename'
 * 2.) Invoke TeX with 'filename' as argument
 * 3.) Check error code from TeX.
 * 4.) If error, move cursor to line #
 * 5.) If no error, invoke the TeX previewer
 *
 */
OPTIONS RESULTS /* return strings are ok to send back to ARexx */
OPTIONS FAILAT 20
'STATUS F' /* return file name to ARexx from QED */
fname=result
'WRITE' /* Save file */
error=rc
/* check for error returns - a NO CHANGES #18 error is OK */
IF error == 0 | error == 18 THEN DO
    /* add '.tex' extension if missing */
    scan=INDEX(fname, ".tex")
    IF scan > 0 THEN nname=fname
    ELSE nname=fname'.tex'
    /* Start AmigaTeX if it hasn't been started already.
    We determine this by looking for its ARexx port.
    If not found, we start TeX, and wait for the ARexx port.
    Plus we do a check to make sure "WaitForPort" found the
    port, or simply timed out.
```

¹ AmigaTeX is a product of Radical Eye software

```

*/
IF SHOW(PORTS,'AmigaTeX') = 0 THEN DO
    ADDRESS COMMAND "NEWSHELL con:////TeX FROM s:texscript"
    ADDRESS COMMAND "WaitForPort AmigaTeX"
    END
IF SHOW(PORTS,'AmigaTeX') THEN DO
    /* Now we send a message to TeX telling it which file to
    process, and a second telling it to come back to us
    at the next TeX prompt. We then check for an error
    condition by sending it the 'ErrorLoc' command.
    TeX sets an ARexx global variable in this case
    formally referred to as an ARexx 'cliplist'.
    */
    'STATUS P'
    path=result
    ADDRESS 'AmigaTeX' 'CD 'result
    ADDRESS 'AmigaTeX' 'TeXify 'nname
    ADDRESS 'AmigaTeX' 'NextPrompt'
    ADDRESS 'AmigaTeX' 'ErrorLoc'
    texerror=GETCLIP('AmigaTeX.ErrorLoc')
    /* We use an ARexx function here to parse the string
    returned by TeX into a filename, line position,
    and character position. TeX returns all 3 values
    as a single string.
    We then use the info to determine if an error occurred, and
    if so, we abort the TeX job, and determine if the
    'filename' error matches the name of the file we
    submitted to TeX for processing. If so, we use the
    line number, and character positioning values to
    position the cursor in QED, and wrap the whole
    thing up with a QED error message.
    If everything went well, we start TeX's previewer,
    and display a short QED message.
    */
    PARSE var texerror filename linenum charpos
    IF filename = "" THEN DO
        ADDRESS 'AmigaTeX' 'Abort'
        IF nname = filename THEN DO
            'GOTO 'linenum' 'charpos
        END
        'ERROR TeX Error'
    END
    ELSE DO
        scan=INDEX(nname,".tex")
        fname=LEFT(nname,scan-1)
        ADDRESS COMMAND 'Preview 'fname
    END
END
END
ELSE DO
    'ERROR Could not find TeX'
END
END
ELSE DO

```

```
'ERROR File Save Error'  
END
```

If I didn't have ARexx, I would have to manually had to do a Save File in the editor, then type "TEX ARexxnotes" then type "PREVIEW ARexxnotes". With ARexx, all this can be done with the press of a key.

However, the user can only make use of this if you the developer implement your program so as to allow ARexx to control it.

1 Basic Design Considerations

First and foremost, ARexx is a string parser. It manipulates strings of characters, hands them off to programs, and awaits results. And therein lies the key to seamlessly implementing ARexx in your programs. Your program has to be expecting string input as commands.

This is somewhat in contravention of the icon-based, point-and-click metaphor that is used by many of the graphic presentation systems such as the Amiga. It is generally easier to graft a GUI2 onto a command oriented program than the other way around.

Let us assume that you are designing a text editor. In general, the first thing that one does is decide what sort of functions the editor will allow. For example:

1. Mark a block (only one block allowed)
2. Cut a marked block to the clipboard
3. Insert a block from the clipboard into the text
4. Save a file to disk
5. Open a new file and read it into the buffer
6. Jump to a particular line
7. Search for a string
8. Search for a string and replace it with another string.

That seems like the beginning of an editor.

Next would be to design the commands that cause these things to happen. This necessitates that both a Syntax and a Semantics be defined. For example, the Syntax rules might be, "The action comes first, is always in capitals, and is followed by a space. Only as many letters of the command as is required for identity need be entered. All characters after that point are ignored." That's a syntax and is the definition of how to create a valid command.

Semantics is the definition of how to figure out what the corectly constructed command means. For instance, "If a SAVE command is entered without a file name, use the same file name as the input filename." With this in mind, we might come up with the following command set:

2 GUI: Graphical User Interface

1. MARKBLOCK [BEGIN — END]
2. CUT
3. INSERT
4. SAVE <filename>
5. OPEN filename
6. JUMP linenumber
7. SEARCH "string to search for"
8. REPLACE "search string" "replacement string"

We would now write a command parser that will take commands in this format and call the proper routines to do the work. For example, we might call this in the following fashion:

Listing 2 Example Command Dispatcher

```
Dispatch(char *InputCommand)
{
    switch (InputCommand[0])
    {
        case 'M':
            return (MarkBlock (GetSecondWord (InputCommand)));
        case 'C':
            return (CutBlockToClipboard());
        case 'I':
            return (InsertBlock());
        case 'O':
            return (OpenFile (GetSecondWord (InputCommand)));
        case 'J':
            return (JumpToLine (GetSecondWord (InputCommand)));
        case 'R':
            return Replace (GetSecondWord (InputCommand));
        case 'S':
            if (InputLine[1] == 'E')
                return (SearchFor (GetSecondWord (InputCommand)));
            else
                return (SaveFile (GetSecondWord (InputCommand)));
    }
}
```

This takes a command string in, interprets it according to the semantic rules that we have determined, and performs the functions necessary to that command.

In front of this sits the two front ends - the menu front end and the ARexx front end. For the ARexx front end, handling the ARexx messages that will be coming in. The code for the ARexx front end would look something like:

Listing 3 ARexx Front End Example Code

```
/* Assume that the message is received in an outer even processing
loop, and that we merely need to parse it in this routine.
*/
ParseARexxMsg(struct REXXMsg *TheMessage)
{
    char *CommandString;
    int ArgCounter;
    int StringSize;
    if(TheMessage->rm_Action != RXCOMM)
    {
        TheMessage->rm_Result1 = 10; /* Serious error */
        TheMessage->rm_Result2 = 100; /* "Invalid command packet"*/
    }
    else
    {
        StringSize = 0;
        for(ArgCounter=0;ArgCounter<16;ArgCounter++)
            if(TheMessage->rm_Args[ArgCounter])
                StringSize+=
                    strlen(TheMessage->rm_Args[ArgCounter]);
        if(CommandString =
            AllocMem((ULONG)StringSize+1L, MEMF_PUBLIC))
        {
            CommandString[0] = '0';
            for(ArgCounter=0;ArgCounter<16;ArgCounter++)
                strcat(CommandString, TheMessage->rm_Args[ArgCounter]);
        }
        TheMessage->rm_Return1 = Dispatcher(CommandString);
        /* It is assumed that the message handler front end will
        ReplyMsg() for me.
        */
    }
}
```

Note that this is not the only way to process the messages, but is intended as an example only.

The basic idea is to get the various strings that are passed to you by ARexx and then to send them through your command parser. Return codes (i.e. error codes) are passed back in the `rm_Return2` field. the SEVERITY of the error is returned in the `rm_Return1` field. After filling in these two fields a normal `ReplyMsg()` will send it back to the main ARexx server.

In order to keep things tidy, it is generally a good idea to have the menu picks also create command strings and send those command strings through the parser as well, thus creating a single point of control for command handling. Generally I handle this through expanding the `MenuItem` structure to include the command string that will be sent to the command dispatcher.

2 Initiating ARexx commands from your program

Your program doesn't need to only accept commands from ARexx, it can send commands to ARexx as well. This generally is done in one of two ways.

The first way that commands are sent to ARexx is generally to specify the name of an ARexx program to be run. For instance, in the instance of the editor/TeX interactions, the editor actually tells ARexx, "Execute the file TeX.QED." ARexx then goes out and takes the contents of TeX.QED and executes it as an ARexx program (The program is listed at the end of these notes as an example).

In order to send the name of an ARexx program file to ARexx, you fill out a RexxMsg structure. Several utilities are available to you to make this easier, much as CreatePort() is available to you to make creating message ports easier.

A RexxMsg looks like the following structure³

Listing 4 RexxMsg structure

```
struct RexxMsg {
    APTR rm_Task;
    APTR rm_LibBase;
    LONG rm_Action;
    LONG rm_Result1;
    LONG rm_Result2;
    STRPTR rm_Args[16];
    struct MsgPort *rm_PassPort;
    STRPTR rm_CommAddr;
    STRPTR rm_FileExt;
    LONG rm_Stdin;
    LONG rm_Stdout;
    LONG rm_Avail;
}
```

Your program needs to fill in the following fields:

1. rm_Action
2. rm_Args

All the other necessary fields will be filled in for you if you use the CreateRexxMsg() function.

The rm_Action field should be initialized with the value RXCOMM in order to start ARexx processing a command. If you want to receive the result string that the program leaves you, then OR in the value

³ Taken from the ARexx include file STORAGE.H included with the original release of ARexx by Bill Hawes

RXFB_RESULT into the rm_Action field. Furthermore, if you are passing just one string in rm_Args[0], even though that string contains various commands, then you will probably also want to OR in the value RXFB_TOKEN to cause ARexx to re-parse the command string for you before passing it to the invoked command file.

Note, however, that the values that are placed in the rm_Args fields are NOT string pointers, but rather pointers to the middle of RexxArg structures. These RexxArg structures are of the following form:

* * *

Listing 5
RexxArg Structure

```
struct RexxArg {  
    LONG ra_Size;  
    UWORD ra_Length;  
    UBYTE ra_Flags;  
    UBYTE ra_Hash;  
    BYTE ra_Buff[8];  
}
```

and are most easily created using the CreateArgstring() procedure. Note that the rm_Args[] pointers actually point to the ra_Buff field of the RexxArg structure, and that although it is declared as a BYTE array of 8 values, it actually can be as long as you require for your string.

The second way that a program can cause ARexx to do something is to send to ARexx the complete "program" in the message to ARexx. This form is called a "string file". While not much is said about these string files in the "ARexx User's Reference Manual" (and even less said in the Commodore Documentation that comes with the Amiga 3000), it is a fairly simple concept. The rx_ARG[0] field points to a string buffer that contains the entire ARexx program.

Such a string file should contain semi-colons in order to separate the individual commands. Also, care should be taken about the quoting conventions of ARexx in order to maintain the proper quoting of parameters.

For example, here is some code from a MAKE utility that uses ARexx as its scripting language as opposed to the UNIX Bourne Shell:

Listing 6
Sending a String File to ARexx

```

doshell(comstring)
char *comstring;
{
    struct RexxMsg *theRexxMsg, *CreateRexxMsg();
    int i;
    char *TempBuffer;
    ULONG BufferLen;
    if(theRexxMsg = CreateRexxMsg(MyPort, "", "COMMAND"))
    {
        BufferLen = strlen(comstring) + 5L;
        if ( !(TempBuffer = (char *)AllocMem(BufferLen, MEMF_PUBLIC |
            MEMF_CLEAR)))
        {
            DeleteRexxMsg(theRexxMsg);
            fatal("Could not allocate room for expansion");
        }
        *((ULONG *)TempBuffer) = BufferLen;
        TempBuffer += 4;
        strcat (TempBuffer, comstring);
        theRexxMsg->rm_Args[0] = TempBuffer;
        TempBuffer -= 4;
        for (i = 1; i < 16; i++)
            theRexxMsg->rm_Args[i] = 0L;
        if(FillRexxMsg(theRexxMsg, 1, 0x0))
        {
            theRexxMsg->rm_Action |= RXCOMM | RXFF_STRING;
            PutMsg(REXXPort, (struct Message *)theRexxMsg);
        }
        else
        {
            FreeMem(TempBuffer, *((ULONG *)TempBuffer));
            DeleteRexxMsg(theRexxMsg);
            fatal("Could not fill rexx message");
        }
    }
    return (await(MyPort,theRexxMsg, TempBuffer) );
}

```

The string file is sent to ARexx in the same manner as the name of an ARexx program is sent, with the addition that RXFF_STRING is ORed into the rm.Action field of the RexxMsg structure. The return codes are managed in the same fashion as elaborated previously.

3 Other ARexx interactions

There are other interactions that you might want to have with ARexx. For example, there may be times that you want to put something into, or fetch something from the ARexx clip list⁴. To access these, you use the various other values for the `rm.Action` field in the `RexxMsg` structure, and send a request off to the ARexx server.

There's a lot more to ARexx than can be said in a one hour discussion. Carefully read the available documentation that came in the "ARexx Reference Manual", and ask questions. You can contact me at k.klop on BIX, K.KLOP on GENIE, or kevin@cbmvax on USENET if you have questions.

⁴ The clip list is like a large pool of string variables that ARexx maintains.

1

2

3



The Amiga AppShell

The information presented here applies to the first public release of AppShell (Alpha 10). This software will only operate under AmigaOS 2.0. Subsequent revisions will contain updated material and any changes will be shown in the change section at the end of this set.

Disclaimer: This is an *alpha* release. CBM reserves the right to violate object code compatibility and even source code compatibility. Interfaces and data structure specifications are subject to change based on the input received from this early release.

This Alpha release is for the Developer's Conference only. Please restrict any discussion to the appropriate channels, specifically, the amiga.com section of BIX. For comments, use:

BIX: dbaker (currently)
Usenet: cbmvax!davidj
US Mail: David N. Junod
Commodore Business Machines, Inc.
1200 Wilson Drive
W.Chester, PA 19380

CONTENTS

Part1: The Standard Amiga User Interface and the Amiga AppShell

This document gives a non-technical overview of the standard Amiga application requirements and how the Amiga AppShell provides an easy way to abide by those requirements.

Part2: Amiga AppShell Implementation Notes

This document details how to develop applications using the AppShell.

Part3: AppShell AutoDoc

Documents extracted from the source code of the AppShell.

The Standard Amiga User Interface and the Amiga Appshell

by David Junod

The purpose of this document is to describe the requirements of a standard Amiga application user interface and how the Amiga AppShell will ease the developer into those requirements.

Scope

This document was developed to provide the following information:

- ☐ Benefits of a Standard User Interface
- ☐ Overview of the Amiga AppShell
- ☐ Elements of the Standard Amiga User Interface

Benefits of a Standard User Interface

Consistency

Consistency in a user interface allows the user to apply previously learned knowledge to each new application. The user will spend less time on figuring out how to get work done, and is therefore more productive.

Coexistence

By following the recommended standard procedures, applications automatically acquire the ability to inter-operate, thereby increasing each application's value.

Overview of the AppShell

The requirements of a standard Amiga application can be overwhelming to a new developer. The initial learning curve to develop on the Amiga is already quite steep, without the additional burden of implementing standard features.

In order to help overcome this initial brick wall, the Amiga *AppShell* was developed. It provides an overall philosophy on programming applications, as well as providing the mechanisms required to implement the standard Amiga requirements.

Philosophy

The AppShell is designed with the following philosophy in mind:

- ❑ The application's functions must be separate from any particular user interface. This allows an application to be controlled in a variety of ways; such as by the mouse, keyboard, or a scripting language, or even another process.
- ❑ The user must be able to personalize the application to fit his or her work habits. This could be as simple as setting the colors and font, or as complex as adding menu items for often used scripts. Remember, the user is trying to get work done and shouldn't be handicapped by any other persons' ideals or quirks.

Implementation

The AppShell has been implemented as a combination of a link library and a shared system library.

The use of a shared system library has a number of benefits:

- ❑ *Application size is greatly reduced* - Common code and data all reside in the library.
- ❑ *Memory requirements are greatly reduced* - Common code and data is only loaded once into memory, regardless of the number of applications running.
- ❑ *Efficient upgrade capabilities* - By installing a new library, all applications which use it will also be updated.

Elements of the Standard Amiga User Interface

The section gives basic descriptions of the components required in a standard Amiga application.

The major components are:

- ☐ Functions (standard and application specific)
- ☐ Graphical User Interface
- ☐ Command Interface
- ☐ Workbench Interface
- ☐ Preferences
- ☐ Miscellaneous

Note that many of the components are user-configurable; these descriptions detail only the default aspects of the components.

Functions

There are basically two types of functions: application functions and standard functions. Application functions are those functions that are unique to the application and standard functions are those that provide basic functions that are common to all applications.

Application-Specific Functions

Functions are what applications are built upon. For example, a text editor has functions to move the cursor up and down, accept keystrokes, and load and save a document.

Standard Functions

The AppShell provides many standard functions. These functions fall into two basic categories:

Standard Internal Application Functions - Information on the internal application functions is beyond the scope of this document and can be obtained by reading the *Amiga AppShell Implementation Notes*.

Standard User-Accessible Functions - These would involve functions that are accessible by the user of the application. One example is the AppShell's ability to learn a sequence of events and save those events into a macro file.

Standard User Accessible Functions

This section details the standard user accessible functions that the AppShell provides. Some of the functions are dependant on certain aspects of a particular user interface; for instance, there wouldn't be any control over window operations if the application didn't have a window.

These functions, and their actions, should be duplicated by all future Amiga applications.

NAME	DESCRIPTION
ACTIVATE	Activate the default or named window. Activation of a window involves de-'iconifying' it, bringing its screen to the front, sizing it to its normal size and making it the active window.
ALIAS	Allows a functions and its parameters to be bound to another name.
BUTTON	Allows the named button's function and parameters to be edited.
CMDSHELL	Opens a command shell whereby the user can interact directly with the application at a command level. Allows the user quick access to functions or macros that they may use so infrequently that they don't desire to bind them to a key, menu or button.
CLIP	Save the selected block or clip area to the current clip file. A name can be specified.
DEACTIVATE	Restores the state that the window and screen was in before the ACTIVATE function.
DISABLE	Disable any particular object of the application. Could be a function, menu item, button, or other object.
EDIT	Provides access to modifiable aspects of the application.
ENABLE	Enable any particular object of the application. Could be a function, menu item, button, or other object.
EXECMACRO	Execute the macro that is currently in memory. It could be a macro that was learned or loaded. Mainly to be used for transient learned macros, more permanent macros would actually be bound to a menu item or gadget.
FAULT	Return the error message for the given error number.
GET	Obtain information on the aspects of an object. Such as what node is currently selected in a Scrolling List gadget's list of items.
GROUP	Allows like objects to be linked together. For example, an APPEDIT group could consist of UNDO, MARK, CUT, COPY, PASTE, and ERASE. Then, whenever such editing isn't available to user, the group could be disabled by with one command.

HELP	Provides access to information on different aspects of the application. Information such as the supported functions, parameters required for functions, and even user notes is readily available.
HOTKEY	Allows access to what function a particular key triggers.
LEARN	Place the application in a state whereby it remembers each action that the user performs and builds a macro containing those actions.
LOADMACRO	Will load the named macro into memory for easy access.
MENU	Provides a means of adding, editing, or deleting a menu item. Allows access to what function that menu item triggers.
PRIORITY	Provides a means of controlling the priority that the system uses in running the application.
RETRIEVE	Retrieve from the current clip file into the clip area. A name can be specified.
RX	Allows access to ARexx functions that may have otherwise been rerouted by the application. Should only be used if there are name conflicts between an application function and an ARexx function.
SAVEMACRO	Will save the current macro memory to the named file.
SELECT	Provides a way to manipulate which object is to be selected. Mainly used to access multiple projects.
SET	Manipulate aspects of an object.
STATUS	Provides information on the status of an object, whether it is enabled, disabled, opened or closed.
STOP	Stop the named operation. Used for such things as STOP LEARN.
STUB	A function that does absolutely nothing.
TOBACK	Send a window behind other windows. A name can be specified.
TOFRONT	Send a window to the front of the other windows. A name can be specified.
WHY	Return information on what the last error was.
WINDOW	Provides a means to specifically open or close a window. A name can be specified.

Graphical User Interface

This section provides information on Graphical User Interface (GUI) aspect of an application.

Gadgets

Several new gadget standards have been established in AmigaOS 2.0. These include a new visual style, as well as a number of new gadget types.

The new gadgets now appear three-dimensional. By using a 3D appearance, it is possible to convey more information in less space. When a gadget appears to be raised, it is quickly identified as being modifiable. A gadget that is always indented, or pushed in, is used for display purposes only.

The new gadget types are:

<i>Action button</i>	Triggers actions or indicates completion.
<i>Checkbox</i>	Indicate a yes or no condition.
<i>Scrolling list</i>	Allows selection or display of variable length lists of information.
<i>Radio button</i>	Allow a single choice out of a list of choices. Mutually exclusive.
<i>Cycle gadget</i>	Allow a mutually exclusive choice in a minimal amount of space. Used only for selection of a state, should not be used to trigger actions. Shift selection would cycle backwards through the choices.
<i>Palette</i>	Allow selection of a pen color from the current color palette.
<i>Scroller</i>	Slider which is used to scroll information in a display area.
<i>Slider</i>	Analog selection of a choice from a range of choices.
<i>Text entry</i>	Entry of a single line of textual information.
<i>Text display</i>	Display textual information.
<i>Numeric entry</i>	Entry of a single line of numeric information.
<i>Numeric display</i>	Display numeric information.
<i>Icon drop box</i>	Indicate that an area is capable of receiving icons from Workbench.

Keys may be bound to gadgets so that the gadget's action can be controlled from the keyboard. Whenever possible, visual feedback is given when the keys are pressed. Following is a list of the visual effects that bound keys can have on gadgets. All action occurs on the downpress of the key. Note that some of the gadgets make use of the shifted and unshifted state of the key.

Gadget Action

Action button	When the key is pushed, the gadget appear to press in. When the key is released, the gadget appears to come back out.
Checkbox	Toggle the state of the check mark.
Scrolling list Radio button Cycle gadget Palette Scroller Slider	Unshifted would cycle forward through the choices. Shifted would cycle backwards through the choices.
Text entry Numeric entry	Activate the gadget for entry.

Users have the capability to edit the key that is bound to the gadget. They are also provided with the ability to add, edit, and delete buttons that are set aside for user configurability.

The AppShell processes gadget information so that font sensitivity, color sensitivity, and internationalization are possible. The AppShell utilizes GadTools and new Intuition features for the gadgets.

Menus

The AppShell will preprocess menu information so that internationalization is possible. Users also have the capability of adding, editing, and deleting menus and menu items for the application.

Keyboard

The AppShell provides the ability to bind any keystroke or sequence of keystrokes to any particular function or macro.

Standard Tools

Using the ASL shared system library, the application has access to standard file and font requesters.

The AppShell maintains a list of projects, and whenever multiple files are selected from the file requester, they are appended to the project list. The AppShell provides an AppWindow requester which allows the user to add, remove, select, and reorder projects in the project list.

Command Interface

Although many feel that a GUI is the easiest way to run an application, others feel that they need a direct command-oriented interface to the application. The AppShell provides several means to accommodate a command-oriented user interface.

ARexx

ARexx is a scripting language that is common to all AmigaOS version 2.0 systems and is also generally available for all Amiga computers running AmigaOS 1.2 and greater. ARexx provides the user with text-oriented control over an application or group of applications. It can be used to link the base functions of an application together to produce new, more powerful functions.

Applications should support the commands described in the *Standard User Accessible Functions* section.

The ARexx port name should be based on the command name used to invoke the application and should be in all upper case. For example, the fictional application *Super Paint IV*, might have a command name of SPIV, would have an ARexx port name of:

SPIV_AREXX

If the application allows multiple ARexx ports, then the use count should be appended to the base name, giving:

SPIV_AREXX_1

Command Shell

The command shell provides the user with a direct method to access the base functions of the application. It achieves this by supplying a console window in which the user can send and receive textual information. It also provides the ability to directly run ARexx scripts without having to specifically bind the script to a particular user interface component.

Keyboard

Any function or macro can be bound to any particular key or sequence of keys. In addition to normal single-key commands, complex multi-key commands, such as Memacs-style bindings, are possible.

Workbench Interface

AmigaOS 2.0 provides three new features which allow applications to obtain information from Workbench. They are: AppIcon, AppMenu, and AppWindow.

AppIcon

The AppIcon feature offers the application the ability to add an icon to the Workbench window and receive messages when an icon is dropped on it or the user double-clicks it.

The AppShell provides the ability for the application to close down its GUI and appear as an icon on the Workbench. Then, when the user double-clicks the icon, the application's GUI reactivates. The AppShell remembers the state of the GUI, so that when the user reactivates the application, the GUI's state is restored.

By using the AppIcon feature of receiving messages when icons are dropped on it, the AppShell application can receive information on additional projects to manipulate. For example, a print spooler application could have a representative icon on the Workbench screen and whenever an icon was dropped into it, the spooler would add the file to the print queue. Whenever the user double-clicks the icons, the print spooler would open up a window showing the contents of the queue and would allow the user to manipulate it.

AppMenu

The AppMenu feature allows the user to add applications to the Workbench tool menu. The AppShell does not provide any additional support for this User feature.

AppWindow

AmigaOS 2.0 allows applications to register with Workbench that they are capable of handling icons that are dropped into their window. For example, the user can pick up an icon of a document and drop it into the window of a word processor, or the user could select a number of ILBM icons and drop them into an application to be made into a slide show.

The AppShell maintains a list of projects, and whenever multiple projects are dropped into the application they are appended to the project list.

Preferences

The AppShell provides the application with the functions necessary to accommodate the user's preferences. It provides the ability to read and write preference files, as well as translate the appropriate aspects of the user interface to utilize those preferences.

In the case where there is no user preference for the application, the AppShell will default to the appropriate Workbench Preference.

Application preferences which don't require notifying the application, should be stored in a directory named *config*, which should be in the home directory of the application.

The naming convention is *<application name>_<type>.<file type>* where:

<i><application name></i>	The same as the command name used to invoke the application.
<i><type></i>	The type of the preference file.
<i><file type></i>	Indicate the format that the file is stored as.

For example, an application called *Super Paint IV* might may have a command name of *SPIV* and would store the following files:

<i>SP-IV_palette.ilbm</i>	For the default color palette which is stored as an ILBM IFF file.
<i>SPIV_pointer.ilbm</i>	For the default pointer sprite which is stored as an ILBM IFF file.

Notice that the *<type>* and *<file type>* reflect the names used by the Workbench preference files.

Preference files that do require notifying the application, should be saved in ENV: in a directory with the same name as the command name for the application. The preference file names should match the naming conventions of non-notification preference files. Permanent storage for these preference files would be in ENVARC: using the same naming conventions.

Miscellaneous

The following is a list of standard features that do not fall into any particular category of their own.

Help

Help should consist of an asynchronous text/image display process which communicates with the application to provide context-sensitive help, which is help based on the current state or mode of the application. Whenever the user presses the HELP key, the application sends an appropriate ID to the help process. The help process would then be activated and would display the corresponding information. The user could then either go through more help screens or continue on in the application.

The AppShell provides all the capabilities and mechanisms required to give the user a context sensitive help system. The AppShell maintains ID's for every component within the application and provides a means for the user to inquire for more information on any one of them. Help information is contained in customizable text files, so that the user can enhance the information with their own notes

See the *Standard User Accessible Functions* section for more information.

Internationalization

The AppShell requires an application to construct a default text array from which it will derive all of its textual information. The AppShell provides a way to utilize the localization library to translate the text into the users' preferred language.

Notification

This feature provides a means for an application function to be activated whenever a file gets modified in any way.

For example, Workbench asks to be notified when the Preference files are modified. If the Workbench pattern file is modified, Workbench is notified and updates the windows with the new pattern.

Tools

The Tool feature provides a method of launching asynchronous processes within an application's environment. These tools could be such things as a calculator, calendar, clock, help system, mouse coordinate display, magnifying glass, or even another copy, or clone, of the main application itself.

Using the tools mechanism, the AppShell makes it possible to clone the application. Another copy of the application will be launched, using the same settings as the original application. Now, multiple projects can be easily started and maintained.

Tools can either be internal or external to the application. External tools are made available as shared system libraries in a sub-directory of libs: named *tools*. For more information on tools, consult the *Amiga AppShell Implementation Notes*.

Simple Interprocess Communication

Due to the internal nature of the AppShell function dispatcher, a simple interprocess communication protocol was necessary. This protocol needed the following features:

- Speed* High speed communication was a necessity due to the amount of information being processed. In order to achieve this, it utilizes standard numeric ID's and a direct bidirectional link with any other process.
- Independence* The ability to communicate must not be dependant on anything other than Exec and the AppShell.

Using this protocol, the AppShell communicates with tools and cloned AppShell applications. This could be as simple as telling another AppShell application to activate or as complex as constantly feeding data on a continually changing process.



Implementation of the Amiga AppShell

by David Junod

The purpose of this document is to describe, in technical detail, the construction of an AppShell application.

Scope

This document was written to provide the following information:

- ☐ Overview of the AppShell Components
- ☐ Technical Breakdown of the Components
- ☐ Overview of the Standard Message Handlers
- ☐ Implementation of a Custom Message Handler

Overview of the AppShell Components

The Amiga AppShell provides the developer of Amiga applications with the ability to easily incorporate a standard, consistent interface for the user.

The AppShell operates under the theory that applications can be broken into several separable modules:

- | | |
|--|--|
| <input type="checkbox"/> User Interface | The means by which the user controls an application. |
| <input type="checkbox"/> Application Functions | Functions specific to the application. |
| <input type="checkbox"/> Standard Functions | Functions that are common to almost all applications, to provide a consistent, solid set of standard operations. |

Of the three, the developer still has to define the first two. But the AppShell provides the third and makes the other two much easier.

Note: Unless otherwise stated, all references to the word "function" refers to either an Application Function or a Standard Function.

The following is a basic breakdown of the major components of the AppShell.

- activate existing application
- open required libraries
- startup parameter parsing
 - locate configuration files
 - Workbench
 - Shell
- initialize message handlers (honor preferences)
 - ARexx
 - Command Shell
 - Commodities
 - IDCMP (Intuition and GadTools)
 - Simple IPC
 - Asynchronous Tools
 - Workbench messages (AppIcon, AppMenu & AppWindow)
 - Others...
- handle messages (event processor) until quit
- shut down message handlers
- close libraries

The base initialization and library manipulation routines are within a link library. The remaining routines reside in a shared system library.

Due to the Alpha state that this software is currently in, *some of the features are not yet fully implemented.*

User Interface Description

The AppShell relies heavily on tags and structure arrays to describe the components of the user interface. All tag and structure arrays are unmodified by the AppShell. This allows any tag array to be used by different processes. Similarly, the entire AppShell is reentrant.

Note that not all of the data elements are copied. An effort has been made to indicate whether an element is copied or just its pointer is used.

Function Table

Every AppShell application has a function table. This table contains information on all the application functions, as well as the standard functions provided by each message handler. Every command to the application is dispatched by the AppShell function dispatcher to the correct function.

Each message handler can add standard functions to the function table. This ensures that each application has a standard base of functions.

Each function has a short name, which is used to bind the function to any particular user interface. A function could be bound to a menu item, a gadget, invoked from a script, or any other user interface.

This short name cannot conflict with any of AppShell's internal names (function or message handler names) or with any of the ARexx keywords.

Functions can have two different states:

<i>Disabled</i>	The function is not able to be invoked. Causes the dispatcher to return a failure-level error.
<i>Enabled</i>	The function is capable of being invoked. The default state of a function.

A function is disabled by sending the DISABLE command, followed by the function name:

```
DISABLE <function>
```

A function is enabled by sending the ENABLE command:

```
ENABLE <function>
```

If the function is bound to any particular graphical user interface item, then that item is also visually disabled. For example, a menu or gadget would be ghosted.

Names

Just about every AppShell item (functions, message handlers, and message handler objects) has a unique name. This allows the application to specify what is to be manipulated.

For example, to disable an application's entire Intuition interface while an ARexx script is running, the following command would be issued:

```
DISABLE IDCMP
```

For greater control, groups can be built by specifying the names of each member of the group. Then, any command to the group would affect all its members.

For example, to set up an edit group:

```
EDIT GROUP APPEDIT UNDO MARK CUT COPY PASTE ERASE
```

This would establish the APPEDIT group with UNDO, MARK, CUT, COPY, PASTE, and ERASE as its members.

To disable a group:

```
DISABLE APPEDIT
```

This would disable all the members of the APPEDIT group, which would include their function table entries and their GUI items (such as their menu items).

Preferences

User definable attributes of an application are called preferences. The AppShell provides the mechanism to read user preferences and transform the objects of the user interface to comply with those preferences. For example, if there is a GUI present in your application, then the AppShell allows the user to specify the font and color palette (as well as a number of other attributes).

The Workbench preference editors are used whenever appropriate and therefore, their names are used also. For a list of the Workbench preference formats that the AppShell supports, see the *IDCMP Message Handler* section.

If the user wants to use different preferences for any particular application, then he would place the altered file into the application's preference directory. Permanent preference files would be placed in a directory *config*, in the home directory of the application.

Preferences also involves localization. The AppShell has been designed to make use of the localization library when it is present.

Technical Breakdown of AppShell Components

Application Variables

```
/* common application data */
struct AppInfo
{
    /* control information */
    STRPTR ai_TextRtn;    /* text return string */
    LONG ai_Pri_Ret;      /* primary error (severity) */
    LONG ai_Sec_Ret;      /* secondary error (actual) */
    BOOL ai_Done;         /* done with main loop? */

    /* base application information */
    BPTR ai_ProgDir;       /* application base directory */
    BPTR ai_ConfigDir;     /* application preference file directory */
    STRPTR ai_ProgName;    /* application base name */
    struct DiskObject *ai_ProgDO; /* application tool icon */

    /* project information */
    struct List ai_ProjList; /* list of projects */
    struct ProjNode *ai_CurProj; /* current project node */
    LONG ai_NumProjs;       /* number of projects in the list */

    /* application information */
    VOID *ai_UserData;     /* UserData */

    /* READ ONLY Intuition-specific information */
    struct TextFont *ai_Font; /* font for screen */
    struct Screen *ai_Screen; /* active screen */
    struct Window *ai_Window; /* active window */
    struct MHOBJECT *ai_CurObj; /* active object (gadget usually) */
    struct DrawInfo *ai_DI; /* Intuition DrawInfo */
    VOID *ai_VI;           /* Gadtools visualInfo */
    WORD ai_MouseX;        /* position at last IDCMP message */
    WORD ai_MouseY;        /* position at last IDCMP message */
    UWORD ai_TopLine;      /* top line */

    /* AppShell-maintained fields */
    /* the remainder of the fields are private to the AppShell */
};
```

The AppInfo structure contains the following components:

The following fields can be accessed by message handlers or functions and are used for control purposes.:

ai_TextRtn	Used by message handler initialization routines to return a string to indicate an error. Used by functions to return text-based information when there isn't an error.
ai_Pri_Ret	Used by functions to return an error level.
ai_Sec_Ret	Used by functions to return an error information. Set to the index of the error in the error text array.
ai_Done	Set to TRUE by the application's Quit function, to indicate to the event processor that the user wants to shut the application down.

The following fields are for obtaining information on the application itself:

ai_ProgDir	Lock on the directory that the application resides in.
ai_ConfigDir	Lock on the directory that contains the preference files for the application.
ai_ProgName	Base name of the application. This name is used to prefix preference files, base name of public message ports and other public names for the application.
ai_ProgDO	A pointer to the Workbench icon used to represent the application.

The following fields are for tracking project related information:

ai_ProjList	Exec list of projects. Each project is represented by a project node (represented by the ProjNode structure).
ai_CurProj	Current project node (represented by the ProjNode structure).
ai_NumProjs	The number of projects in the project list.

The following field is for application specific data:

ai_UserData	Application specific user data.
-------------	---------------------------------

The following fields are maintained by the IDCMP message handler and are READ ONLY. Some of the more timely items are only fresh when the function was triggered by an IDCMP message.

ai_TextFont	Font used for text items inside the window.
ai_Screen	Screen for all the applications' windows.
ai_Window	Pointer to the window that was active when the function was invoked.
ai_CurObj	GUI item that triggered the function.
ai_DI	DrawInfo for the applications screen (used by Intuition).
ai_VI	VisualInfo for the applications screen (used by GadTools).
ai_MouseX	ai_MouseY Coordinates of the mouse when the function was invoked.
ai_TopLine	The first usable line in a window (based on window title bar size plus one).

Activation

Activation is used when you only want one instance of your application running at any one time. To achieve activation using AppShell, the application has to have a single Simple InterProcess Communication (SIPC) port.

If the user attempts to load another copy of the application, then the original application gets an Activation message through its SIPC port. This causes the application to come to the front, opened and activated.

The tags required for activation are:

```
/* Simple IPC message handler initialization tags */
struct TagItem handle_SIPC[] =
{
    {APSH_Setup,    setup_sipcA},                /* SIPC set up routine */
    {APSH_Status,  P_ACTIVE | P_SINGLE},        /* make active and only */
    /* allow one instance */
    {APSH_Rating,  REQUIRED},                    /* application requires it */
    {TAG_DONE,}
};
```

The tags used in this example are:

APSH_Setup	Pointer to the routine used to initialize the message handler system. In this case, it is the Simple Interprocess Communications message handler.
APSH_Status	In this case the message handler would be made active at initialization time and only one port of the given name is allowed.
APSH_Rating	Used to indicate if the message handler is optional or required. The AppShell will shut down if the message handler is required and can't be initialized.

Library Handling

The AppShell provides ways to open and close a list of shared system libraries. Libraries can be marked as optional or required. AppShell will shut down if the library is required and can't be opened.

An example of the data and tags required for maintaining a number of libraries would be:

```
/* library bases */
extern ULONG SysBase;           /* required for pragmas */
ULONG IntuitionBase;           /* intuition processing */
ULONG GadToolsBase;            /* gadtools processing */
ULONG WorkbenchBase;           /* AppWindow processing */

/* list of libraries for our application */
struct TagItem libs[] =
{
    {APSH_LibVersion, 36L},      /* minimum version */
    {APSH_LibStatus,  REQUIRED},  /* we have to have them... */
    {APSH_LibNameTag,  APSH_Intuition},
    {APSH_LibBase,    &IntuitionBase},
    {APSH_LibNameTag,  APSH_Gfx},
    {APSH_LibBase,    &GfxBase},
    {APSH_LibNameTag,  APSH_Workbench},
    {APSH_LibStatus,  OPTIONAL}, /* not required to run */
    {APSH_LibBase,    &WorkbenchBase},
    {TAG_DONE,}
};
```

The tags used in this example are:

APSH_LibVersion	Used to indicate the minimum library version required. This version level is in effect until another APSH_LibVersion is specified.
APSH_LibStatus	Indicate whether the application requires the library or not. REQUIRED or OPTIONAL are the only valid values. This status is in effect until another APSH_LibStatus is specified.
APSH_LibNameTag	A number is used to indicate the library name. See <internal/appshell.h> for a complete listing of valid name tags.
APSH_LibBase	A pointer to the library base variable name. This tag is what triggers a library open, so must be placed after any of the Lib tags mentioned above.

Startup Processing

The AppShell obtains the following application information once the shared system libraries have been opened and before the message handlers have been initialized:

- Home directory
- Preference file directory
- Base name
- Tool icon

This information is obtained whether the application was called from the Shell or from Workbench. If the application was invoked from Workbench, then icon tool type parsing applies. Otherwise, the standard Shell READARGS is used. At this time READARGS has not been implemented in the AppShell. If multiple projects were passed, they are appended to the project list.

Here is a code fragment showing how the startup arguments are passed to the AppShell.

```
/* main loop for our application */
extern struct WBStartup *WBenchMsg;

main (int argc, char **argv)
{
    HandleApp (argc, argv, WBenchMsg, our_app);
}
```

Function Table Entries

All functions that require a user interface are placed in an array called the *Function Table*. The function table is built from entries made up by the Funcs structure.

The Funcs structure contains the variables required for a function table entry. Here is what the Func structure contains:

```
/* function table entry */
struct Funcs
{
    STRPTR fe_Name;          /* Name of function */
    VOID (*fe_Func)(struct AppInfo *, STRPTR, struct TagItem *);
    ULONG fe_ID;             /* ID of function */
    ULONG fe_Key;            /* Hotkey assigned to the function */
    ULONG fe_HelpID;         /* index into text catalogue for help */
    ULONG fe_Flags;          /* Status of function */
    STRPTR fe_Params;        /* Parameters for function */
    ULONG *fe_GroupID;       /* array of group ID's */
};
```

The Funcs structure contains the following components:

fe_Name	A pointer to the short name of the function. This is the name that would be used in bindings or scripts.
fe_Func	Pointer to actual function invoked by the event processor.
fe_ID	Unique number assigned to the function. This is the number used in bindings; such as menu item, gadgets or hot keys.
fe_Key	Hotkey binding for this function. Whenever the key is pressed, the function is dispatched.
fe_HelpID	Index into text catalogue for single line help on function.
fe_Flags	Flags set by the event processor. An application can initialize a function in the disabled state, by setting the FUNC_DISABLED flag in the function table declaration.
fe_Params	When present, these are the actual parameters that would be passed to the function whenever it is invoked. Currently not implemented.
fe_GroupID	Pointer to a non-zero terminated array of group ID's that this function entry belongs in. Currently not implemented.

Here is an example of how an application would set up the function table:

```
/* function ID's */
#define NewID  APSH_USER_ID+1L
#define OpenID APSH_USER_ID+2L
#define SaveID APSH_USER_ID+3L
#define QuitID APSH_USER_ID+4L
#define LAST_ID  APSH_USER_ID+5L

/* function prototypes */
NewFunc (struct AppInfo *, STRPTR, struct TagItem *);
OpenFunc (struct AppInfo *, STRPTR, struct TagItem *);
SaveFunc (struct AppInfo *, STRPTR, struct TagItem *);
QuitFunc (struct AppInfo *, STRPTR, struct TagItem *);

/* function table */
struct Funcs ftable[] =
{
    {"NEW", NewFunc, NewID, NewHID},
    {"OPEN", OpenFunc, OpenID, OpenHID},
    {"SAVE", SaveFunc, SaveID, SaveHID},
    {"QUIT", QuitFunc, QuitID, QuitHID},
    {NULL, NO_FUNCTION,}, /* end of function table */
};
```

Note that the last NO_FUNCTION is required to mark the end of the function table.

The application function ID's must start at the predefined (defined in *<internal/appshell.h>*) value APSH_USER_ID.

Once the function table has been defined, actual application functions need to be established. In order to separate the user interface from the functionality of the application, a common function interface needs to be defined. Necessary information has to be passed to the function regardless of what event triggered the function. Each standard AppShell and Application function has a common argument interface.

```
/* sample function declaration */
StubFunc (ai, line, tl)

struct AppInfo * ai; /* standard application variables */
STRPTR line; /* complete command line */
struct TagItem *tl; /* tag list of attributes */
```

The parameters for the common argument interface are:

- ai - A pointer to the AppInfo structure that contains all the global data required for this application.
- line - The complete command line that the function was invoked by.
- tl - TagItem array of arguments for the function.

The AppShell provides the capability to maintain multiple projects. These projects could be maintained as single threads or multi-threads. A single threaded application, for example, could be a print spooler that steps through the list of projects. A multi-threaded application, for example, could be a text editor that uses a different process for each project.

The AppShell uses the ProjNode structure to maintain information on the projects that the application is working on. Here is what the ProjNode structure looks like:

```
/* information maintained for each project in the project list */
struct ProjNode
{
    struct Node pn_Node; /* embedded Exec node */

    /* AppShell information. Read only for application */
    struct DateStamp pn_Added; /* date stamp when added to list */
    BPTR pn_ProjDir; /* lock on project directory */
    STRPTR pn_ProjPath; /* pointer to the projects' complete name */
    STRPTR pn_ProjName; /* pointer to the projects' name */
    struct DiskObject *pn_DObj; /* pointer to the projects' icon */
    LONG pn_ID; /* user selected order */
    APTR pn_Extens1; /* *** PRIVATE *** SYSTEM USE ONLY */
    /* Application information */
    ULONG pn_Status; /* status of project */
    ULONG pn_ProjID; /* project ID */
    UBYTE pn_Name[32]; /* project name */
    APTR pn_UserData; /* UserData for project */
    BOOL pn_Changed; /* has project been modified? */
    APTR pn_Extens2; /* *** PRIVATE *** SYSTEM USE ONLY */
};
```


The following fields are maintained by the AppShell and are available only as *read only* to the application.

pn_Node	Embedded Exec node.
pn_Added	Date stamp of when the project was added to the project list. Currently not implemented.
pn_ProjDir	Lock on the directory that the project is located in. Maintained by the AppShell, NOT to be unlocked by the application.
pn_ProjPath	Pointer to the complete name, including path, of the project.
pn_ProjName	Pointer to the name of the project.
pn_DObj	Pointer to the icon (DiskObject) for this project.
pn_ID	ID used to sort the project list by user selected order.
pn_Extens1	Reserved for AppShell use only.

The following fields belong to the application.

pn_Status	Status of the project.
pn_ProjId	Project ID.
pn_Name	Name of the project.
pn_UserData	Application specific data.
pn_Changed	Has the project been edited since last save?
pn_Extens2	Reserved for AppShell use only.

Error and Text Handling

The application should define the possible text messages that could be used and build an array of messages. Then whenever a message is needed, it is referenced by the index into the array.

If an error is encountered, the index value for the message is placed in `ai_Sec_Ret`, the error level is placed in `ai_Pri_Ret`, and the text is placed in `ai_TextRtn`. The AppShell will then make sure that user is notified of the error in an appropriate way. For example, if the command that encountered the error was triggered directly by the IDCMP message handler, then a requester will be displayed. Currently, all messages are displayed using an `AutoRequester`.

Example:

```
/* application error table */
STRPTR def_text[] =
{
    "I'm OK",           /* padding */
    "Illegal transformation on %s",
    NULL                /* NULL termination is required */
};

/* easy to remember define */
#define ERROR_TRANSFORM 1

/* sample application tags */
struct TagItem our_app[] =
{
    {APSH_FuncTable, ftable},
    {APSH_DefText,  def_text},
    ...
    {TAG_DONE,}
};

...

/* sample function returning an error */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    STRPTR name;

    ...

    /* sample error return */
    ai->ai_Pri_Ret = RETURN_WARN;
    ai->ai_Sec_Ret = ERROR_TRANSFORM;
    ai->ai_TextRtn = PrepText (ai, APSH_USER_ID, ERROR_TRANSFORM, name);
}
```

The new tags used in this example are:

APSH_DefText Pointer to the default text table for this application.

Message Handler Initialization

The AppShell maintains a list of message handlers. It starts by first calling the initialization routine of each message handler in the list. These message handlers can be marked as optional or required. The AppShell will shut down if a required message handler could not be initialized.

The following is a code fragment that outlines what is necessary to implement an optional ARexx message handler in your application:

```
/* ARexx message handler initialization tags */
struct TagItem handle_AREXX[] =
{
    {APSH_Setup, setup_arexxA},
    {APSH_Extens, (ULONG)"our"},
    {APSH_Status, P_ACTIVE | P_SINGLE},
    {APSH_Rating, OPTIONAL},
    {TAG_DONE,}
};

/* application specification tags */
struct TagItem our_app[] =
{
    {APSH_FuncTable, ftable},
    {APSH_DefText, def_text},
    {APSH_OpenLibraries, libs},
    {APSH_AddHandler, handle_AREXX},
    {TAG_DONE,}
};

/* main loop for our application */
extern struct WBStartup *WBenchMsg;

main (int argc, char **argv)
{
    HandleApp (argc, argv, WBenchMsg, our_app);
}
```

The new tags used in this example are:

APSH_AddHandler	A pointer to the tags for a message handler for this application. There can be multiple instances of this tag in this array.
APSH_Extens	ARexx macro file name extension.
APSH_FuncTable	Pointer to the function table for this application.
APSH_OpenLibraries	A pointer to the library tags for this application.

Once the message handler list has been successfully initialized, the AppShell will go into its event processing mode.

While in the event processing mode, the AppShell will wait on the signal bits of all the registered message handlers. Whenever an event comes in, the appropriate message handler gets called. This process continues until the user indicates that he or she wants to leave the application.

When the application is told to quit, the AppShell enters the shut down mode. While in shut down mode, the AppShell loops through the list of message handlers, calling the shut down routine of each message handler, until the list is empty.

Standard Message Handlers

The following is an overview of each of the message handlers that are provided in the AppShell shared library. Information is given on:

- ☐ Short name of the message handler.
- ☐ List of tags that the message handler can translate.
- ☐ Standard Functions that the message handler implements.
- ☐ Extended low-level functions that the message implements.
- ☐ The preference files that the message handler gets user settings from. Currently not implemented.
- ☐ A short example on how an application can implement the message handler.

Even though it isn't a message handler, information on the AppShell is also given due to the number of standard functions that it provides.

Appshell

The AppShell adds quite a number of standard functions to the function table. It also provides a number of function entry points.

Tags

APSH_AppInit	Function ID to dispatch after the message handlers have been initialized, and before entering the event processing stage.
APSH_AppExit	Function ID to dispatch after ai_Done has been set to TRUE, and before running shut down on the message handlers.
APSH_SIG_C	Function ID to dispatch when the ^C signal is sent to the application. Should be set to the Quit function.
APSH_SIG_D	Function ID to dispatch when the ^D signal is sent to the application.
APSH_SIG_E	Function ID to dispatch when the ^E signal is sent to the application.
APSH_SIG_F	Function ID to dispatch when the ^F signal is sent to the application.

Standard Functions

ALIAS	Link a new command name to an old command name. Partially implemented.
DISABLE <name>	Disable a message handler, function, interface object or group. If there is a GUI item, then it will disable that item also, IE if the function is bound to a menu item, then it disables that menu item. Currently only disables functions and gadgets.
EDIT <name>	Edit the named item. Passes to the appropriate message handler's editing routine. Currently not implemented.
ENABLE <name>	Enable a message handler, function, interface object or group. Currently only enables functions and gadgets.
EXECMACRO	Executes the current macro. Currently not implemented.
FAULT <error>	Returns, in the ai_TextRtn field, the text assigned to <error>. Currently not implemented.
GET <name> [value]	Get [value] from <name>. Useful for getting the attributes of objects. Currently not implemented.
GROUP [EDIT] ...	Edit a group. Currently not implemented.
HELP [function]	HELP alone, will return the functions in the function table. Help <function> will return information for the function. Eventually will bring up a GUI Help system. Currently not implemented.
LEARN [STOP]	Start the macro learn process. Causes the dispatcher to memorize every function that is passed through it. The STOP argument will end the macro learn process. Partially implemented.
LOADMACRO <filename>	Load a macro into memory from the named file. Currently not implemented.
SAVEMACRO <filename>	Save the current macro to the named file. Currently not implemented.
SET <name> [value]	Set <name> to [value]. Useful for setting the attributes of objects. Currently not implemented.
STATUS <name>	Will return information on the named object, such as its current state (Enabled, Busy, or Disabled), use count, parameters, and description. Currently not implemented.
STOP <name>	Stops the named function. Looks for <name> in the function table, and passes STOP to it.
STUB	Function which does nothing.

ARexx Message Handler

The ARexx message handler provides a standard scripting language, as well as string oriented interprocess communications. Currently the ARexx message handler only implements commands. Variable manipulation routines and function host capabilities are planned.

Message Handler Name

AREXX

Tags

APSH_ARexxError	Function ID to dispatch when there is an ARexx command error. Defaults to StubID.
APSH_ARexxOK	Function ID to dispatch when an ARexx command succeeds. Defaults to StubID.
APSH_Extens	ARexx macro file name extension. Defaults to .rexx.
APSH_Port	Base name for the applications public ARexx port. Defaults to the application's base name with _AREXX appended.
APSH_Status	Recognizes: P_ACTIVE Activate the ARexx message handler at initialization time. P_INACTIVE Leave the ARexx message handler inactive Causes the message handler to reply to each incoming message with the return value set to RETURN_WARN. P_SINGLE Use the base port name as is. P_MULTIPLE Append _# to the base port name, where # is incremented until it finds a unused name.

Extended Low-Level Message Handler Functions

AH_SENDCMD	Sends the passed command (APSH_CmdString) to ARexx.
------------	---

Standard Functions

RX	Execute an ARexx command. Only to be used when there is a name conflict between an ARexx command name and a function short name. Currently not implemented.
WHY [GET]	Return more information on the last error. The information is placed in the ai_TextRtn field, and could be text or a number sprintf()ed into text.

In an ARexx script, the primary return value should indicate failure level. This command allows us to get at the real reason a command may have failed.

This function is currently not implemented.

Preferences

<none>

Example

See the *Message Handler Initialization* section for an example of the set up required for an ARexx interface.

Command Shell Message Handler

The *Command Shell Message Handler* provides a command shell for the more advanced user. It allows the user to directly dispatch any of the functions in the function table. It also gives direct access to the power of ARexx without always having to specify the ADDRESS or macro file name extension.

Message Handler Name

DOS

Tags

APSH_CloseMsg	Message to display in the Shell before attempting to close it. Defaults to "Waiting for macro return".
APSH_CMDWindow	Initial Shell window specification. Defaults to "CON:0/150/600/50/Command Shell/CLOSE".
APSH_Prompt	Initial Shell prompt. Defaults to "Cmd>".
APSH_Status	Recognizes: P_ACTIVE Open the Command Shell window at initialization time. P_INACTIVE Keep the Command Shell window closed until specifically requested to open.

Extended Low-Level Message Handler Functions

<none>

Standard Functions

CMDSHELL [OPEN/CLOSE]	OPEN, opens the command shell window. CLOSE, closes it. Defaults to OPEN.
-----------------------	---

Preferences

commandshell.win	Window size and placement preferences, includes ZOOM settings.
------------------	--

Example

```
struct TagItem handle_SHELL[] =
{
    {APSH_Setup,    setup_dosA},
    {APSH_Status,  P_INACTIVE},
    {APSH_Rating,  OPTIONAL},
    {TAG_DONE,}
};
```

IDCMP Message Handler

The IDCMP message handler provides the application with a Graphical User Interface (GUT) through the use of Intuition and GadTools.

Maintains such things as:

- ☐ Public or private screens - Manages the DrawInfo and VisualInfo for the application screen. Also tracks the public screen information whenever appropriate.
- ☐ Multiple windows - Manages an unlimited number of application windows.
- ☐ Translating Workbench preferences - Translate user-interface object description, based on user preferences, into GadTools and Intuition objects. For instance, it will scale the objects according to the font; or remap images according to the color palette. Not completely implemented yet.
- ☐ Snapshotting window preferences - Manages saving/restoring user preferences for individual windows. Includes such things as placement, size, and zoom settings. Currently only saved during execution.
- ☐ GadTool gadgets - Converts object descriptions into GadTools structures and tags.
- ☐ Intuition gadgets - Allows complex user defined gadgets, such as the SketchPad used in the Icon Edit program. Not currently implemented.
- ☐ Custom Intuition gadgets - Supports custom intuition gadget classes. Not currently implemented.
- ☐ User configuration of gadgets - Provides a standard button editing tool. Allows the user to define the text, function, and placement of action gadgets (buttons). Not currently implemented.
- ☐ Nested menus - Allows a function to bring up other menu strips. Not currently implemented.
- ☐ User configuration of menus - Provides a standard menu item editing tool. Allows the user to define the text, function, and placement of menu items. Not currently implemented.

- ❑ Nested HotKeys - Allows complex keyboard commands, like those provided by *emacs* and *vi*. Not currently implemented.
- ❑ User configuration of HotKeys - Provides a standard HotKey editing tool. Allows the user to define a key sequence and its function. Not currently implemented.
- ❑ Images, borders and text - Object management of images, text and common border types.
- ❑ Pointer management - Manages pointer rectangles. For instance, in the SketchPad, while the cursor is over the drawing area, the cursor will reflect the current drawing state. When the cursor is over any other portion of the window, the cursor would be the user's default pointer. Will also load the user's preference in busy pointers. Not currently implemented.

To define GUI items such as buttons, sliders and scrolling lists, we use an array of Object structures. Here is what the Object structure looks like (defined in *<internal/appshell.h>*):

```
/* Intuition user-interface object (gadget, border, text, etc... */
struct Object
{
    struct Object *o_NextObject; /* next object in array */
    ULONG o_Group; /* group that object belongs in */
    ULONG o_Type; /* type */
    ULONG o_ObjectID; /* ID */
    ULONG o_Flags; /* see gadtools defines */
    UBYTE o_Key; /* hotkey */
    STRPTR o_Name; /* name */
    ULONG o_LabelID; /* index of label in the text table */
    struct Rectangle o_Outer; /* size w/label */
    struct TagItem *o_Tags; /* tags for object */
    APTR o_UserData; /* user data for object */
};
```

The Object structure contains the following components:

<code>o_NextObject</code>	Pointer to the next object in the array.
<code>o_Group</code>	The layout group that the object belongs in.
<code>o_Type</code>	Indicates type of objec. The descriptions of the object type are in the next section.
<code>o_ObjectID</code>	Function ID to dispatch when a GadTool event occurs for this object. Only valid for GadTool objects.
<code>o_Flags</code>	Object flags. Currently APSH_OBUF_CLOSEWINDOW is the only valid flags.

<code>o_Key</code>	HotKey binding for this object. Whenever the key is pressed, the <code>o_ObjectID</code> function is dispatched. If possible, that gadget is also visually selected. For example, an action gadget would appear visually depressed as long as the key is pressed. If the key is bound to a slider or scroller, then pressing the key should decrease the value, while SHIFT-key should increase the value. Currently only does text gadgets and action gadgets.
<code>o_Name</code>	All objects are placed into a list for the window that they belong in. <code>o_Name</code> is the name that is used for the list entry (node) for this object. This name should be unique to this window.
<code>o_LabelID</code>	A numeric ID assigned to the visual text label for this object.
<code>o_Outer</code>	The rectangle which describes the placement and size of entire object, including the visual label, if pertinent.
<code>o_UserData</code>	Pointer available to the application for binding data to a particular object. Note that the gadget's <code>UserData</code> field is currently used by the <code>AppShell</code> , and application user data is assigned to the <code>ObjectNode</code> 's <code>UserData</code> field.
<code>o_Tags</code>	Additional parameters for defining the object. A description of the available tags follows the next section.

Following are the valid object types (defined in `<internal/appshell.h>`):

GadTool gadgets

<code>OBJ_Generic</code>	Action gadget.
<code>OBJ_Button</code>	Check box for boolean values.
<code>OBJ_Checkbox</code>	String gadget for numeric entry.
<code>OBJ_Integer</code>	Scrolling list gadget.
<code>OBJ_Listview</code>	Mutual exclusion gadget.
<code>OBJ_MX</code>	Numeric display box.
<code>OBJ_Number</code>	Cycle gadget.
<code>OBJ_Cycle</code>	Color palette gadget.
<code>OBJ_Palette</code>	Scroll gadget.
<code>OBJ_Scroller</code>	Slider gadget.
<code>OBJ_Slider</code>	String gadget for text entry.
<code>OBJ_String</code>	Text display box.
<code>OBJ_Text</code>	

Other gadgets

<code>OBJ_Display</code>	Freeform display box.
<code>OBJ_Select</code>	Freeform action gadget.
<code>OBJ_Dropbox</code>	AppWindow drop box.
<code>OBJ_GImage</code>	Action gadget with a graphic label.

Images

<code>OBJ_Image</code>	Display an image. Could be a scaleable (vector) image.
------------------------	--

Borders

OBJ_Plain	Plain border of detail pen color.
OBJ_BevelIn	Pushed in 3D border.
OBJ_BevelOut	Pushed out 3D border.
OBJ_DblBevelIn	Pushed in 3D embossed border.
OBJ_DblBevelOut	Pushed out 3D embossed border.

Other object types

OBJ_OuterDim	Outer dimensions of window.
OBJ_Screen	Screen dimensions (not supported currently)
OBJ_Window	Window
OBJ_Group	Group information (not supported currently)
OBJ_VFill	Vertical fill area (not supported currently)
OBJ_HFill	Horizontal fill area (not supported currently)

The following are valid AppShell tags for Objects (GadTool tags apply to objects that translate into GadTool gadgets):

APSH_GTFlags	GadTools NewGadget flags to use for this object.
APSH_ObjDown	Function ID to dispatch on the downpress of an Intuition gadget.
APSH_ObjHold	Function ID to dispatch when an Intuition gadget is being held down.
APSH_ObjRelease	Function ID to dispatch on the release of an Intuition gadget.
APSH_ObjDblClick	Function ID to dispatch when an Intuition gadget has been double-clicked.
APSH_ObjAbort	Function ID to dispatch when the right mouse button has been pressed while an Intuition gadget is active.
APSH_ObjAltHit	Function ID to dispatch when an Intuition gadget is selected while holding either ALT key.
APSH_ObjShiftHit	Function ID to dispatch when an Intuition gadget is selected while holding either SHIFT key.
APSH_ObjData	Used to provide the data for the object whenever it is of type OBJ_GImage or OBJ_Image.
APSH_ObjInner	Rectangle describing the placement and size of a scaleable image within an object. Not currently implemented.
APSH_ObjPointer	Used to specify the name of the sprite pointer for the rectangle defined by o_Outer.

Eventually, AppShell will re-layout the objects whenever the object description changes. For instance, language localization can cause text length of buttons or menus to change.

Message Handler Name

IDCMP

Tags

APSH_DefWinFlags	Default window flags.
APSH_GTMenu	GadTools-style NewMenu array.
APSH_HotKeys	Hotkey array used to specify the function binding for any particular key.
APSH_Menu	Intuition-style menu array (obsolete).
APSH_NewScreen	NewScreen structure.
APSH_NameTag	Unique name for any given window environment.
APSH_NewScreenTags	Tags to use when opening the screen.
APSH_NewWindow	NewWindow structure.
APSH_NewWindowTags	Tags to use when opening the window.
APSH_Objects	Array of objects in the Graphical User Interface. Translated into GadTools or Intuition objects, according to user preferences.
APSH_Status	Recognizes: P_ACTIVE Open the Intuition environment immediately. APSH_WindowEnv Information on a single window.

Extended Low-Level Message Handler Functions

<none>

Standard Functions

ACTIVATE [name]	Activates the named window. Activation involves opening the window, bringing it to front and making it the ACTIVE window. Defaults to MAIN.
DEACTIVATE [name]	Return an activated window to its prior inactive state. Not currently implemented.
GADGET [EDIT] ...	Allows the user to bind a function to an action gadget (button). Not currently implemented.
HOTKEY [EDIT] ...	Allows the user to bind a function to a key. Partially implemented.
MENU [EDIT] ...	Allows the user to bind a function to a menu item. Not currently implemented.
TOFRONT [name]	Brings the named window to front.
TOBACK [name]	Sends the named window to back.
WINDOW [name]	Opens the named window. Can also close the window if you append CLOSE to the command.

Preferences

Handling of preference files is not currently implemented. Following are the files that the AppShell will respect.

<i>palette.prefs</i>	Colors to use in custom or public screen.
<i>pointer.prefs</i>	Normal pointer
<i>busypointer.prefs</i>	Busy pointer
<i><name>pointer.prefs</i>	Other pointers, such as mode (fill, text, etc) indicators, whenever appropriate.
<i>screenmode.prefs</i>	Screen mode/type
<i>screenfont.prefs</i>	Screen font, used for menus and window title text.
<i>sysfont.prefs</i>	Font to use inside the windows.
<i>win.pat</i>	Backfill pattern for windows.
<i>wb.pat</i>	Backfill pattern for background window or screen.
<i>printer.prefs</i>	Printer preferences (w/notification)
<i>printergfx.prefs</i>	Graphic printer preferences (w/notification)
<i><window title>.win</i>	Window size and placement preferences, includes ZOOM settings.

This is a complete example on how to do a "Hello World" application.

```
/* Hello.c
 * AppShell application that displays Hello World
 */
#include <internal/appshell.h>
#include <internal/appshell_protos.h>

/* sample function prototypes */
VOID CancelFunc (struct AppInfo *, STRPTR, struct TagItem *);
VOID QuitFunc   (struct AppInfo *, STRPTR, struct TagItem *);

/* sample function ID's */
enum
{
    DUMMYID = APSH_USER_ID,
    OkayID,
    CancelID,
    LAST_ID
};

/* sample function table */
struct Funcs func_table[] =
{
    {"CANCEL", CancelFunc, CancelID,},
    {"QUIT",   QuitFunc,   QuitID,},
    {NULL, NO_FUNCTION,}
};
```

```

/* sample default text table */
STRPTR def_text[] =
{
    NULL,          /* 0 */
    "Sample Window", /* 1 */
    "Hello, World!", /* 2 */
    "Okay",        /* 3 */
    "Cancel",       /* 4 */
    NULL
};

/* sample object array */
struct Object objects[] =
{
    {&objects[1], 0, OBJ_Window, NULL, NULL, NULL, "MAIN", 1L,
     { 1, 1, 0, 0 }, },

    {&objects[2], 1, OBJ_Text, NULL, NULL, NULL, "TEXT", 2L,
     { 10, 14, 304, 14 }, },

    {&objects[3], 2, OBJ_Button, QuitID, APSH_OBJF_CLOSEWINDOW, RETURN,
     "OKAY", 3L, { 10, 32, 60, 12 }, },
    {NULL, 2, OBJ_Button, CancelID, APSH_OBJF_CLOSEWINDOW, ESC,
     "CANCEL", 4L, { 254, 32, 60, 12 }, }
};

/* sample menu array */
struct NewMenu menus[] =
{
    {NM_TITLE, "Project", 0, 0, 0, 0, },
    {NM_ITEM, "Quit", "Q", 0, 0, V ( QuitID ), },
    {NM_END, }
};

extern ULONG SysBase;
ULONG GadToolsBase, GfxBase, IntuitionBase;

/* sample window environment */
struct TagItem window_env[] =
{
    {APSH_NameTag, (ULONG)"MAIN"},
    {APSH_GTMenu, (ULONG)menus},
    {APSH_Objects, (ULONG)objects},
    {TAG_DONE, }
};

```



```

/* sample tags for shared system libraries */
struct TagItem libraries[] =
{
    {APSH_LibVersion,      36L},
    {APSH_LibStatus,      REQUIRED},
    {APSH_LibNameTag,     APSH_GadTools},
    {APSH_LibBase,        &GadToolsBase},
    {APSH_LibNameTag,     APSH_Gfx},
    {APSH_LibBase,        &GfxBase},
    {APSH_LibNameTag,     APSH_Intuition},
    {APSH_LibBase,        &IntuitionBase},
    {TAG_DONE,}
};

/* sample tags for Intuition interface */
struct TagItem handle_IDCMP[] =
{
    {APSH_Setup,          setup_idcmpA},
    {APSH_WindowEnv,      (ULONG>window_env},
    {APSH_Status,         P_ACTIVE},
    {APSH_Rating,         REQUIRED},
    {TAG_DONE,}
};

/* sample main application tag list */
struct TagItem sample_app[] =
{
    {APSH_FuncTable,      (ULONG)func_table},
    {APSH_DefText,        (ULONG)def_text},
    {APSH_OpenLibraries,  (ULONG)libraries},
    {APSH_AddHandler,     (ULONG)handle_IDCMP},
    {TAG_DONE,}
};

/* Workbench startup message */
extern struct WBStartup *WBenchMsg;

/* sample main loop. This is it! */
VOID main (int argc, char **argv)
{
    HandleApp (argc, argv, WBenchMsg, sample_app);
}

/* sample function to dispatch when the cancel button gets pressed */
VOID CancelFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    printf ("Cancel\n");
    PerfFunc (ai, NULL, "QUIT", tl);
}

/* sample Quit function */
VOID QuitFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    printf ("Quit\n");
    ai->ai_Done = TRUE;
}

```

Simple Interprocess Communications Message Handler

The Simple Interprocess Communications Message Handler (SIPC) provides fast and simple communications between applications and tools.

It allows only one invocation of an application to be around. If you click on the icon of an already running application, it will tell the application to come to the front and become active.

The tool message handler uses the SIPC to tell tools and applications to shut down.

Message Handler Name

SIPC

Extended Low-Level Message Handler Functions

<none>

Standard Functions

<none>

Preferences

<none>

Tool Message Handler

The *Tool Message Handler* provides the application with the capability to run asynchronous processes. If an application allows multiple projects, the tool message handler offers a way for the application to start a new process for each project. It also allows tools relevant to the application to be run asynchronously. Such tools could be a calculator, a magnifying glass, or even a clock.

The help system would be a tool that maintains SIPC with the main application.

Message Handler Name

TOOL

Extended Low-Level Message Handler Functions

<none>

Standard Functions

<none>

Preferences

<none>

Implementing a Message Handler

The following information is useful when implementing a message handler.

Each message handler must provide the following low-level message handler functions:

- ☐ Setup Message handler initialization
- ☐ Open Enable the message handler or one of its objects.
- ☐ Handle Translate messages into calls to the AppShell function dispatcher.
- ☐ Close Disable the message handler or one of its objects.
- ☐ Shutdown Shut down the message handler.

One of the base components of message handlers is the MHOBJect structure. This structure contains the fields necessary to maintain nested layers of message handler objects.

```
/* message handler object node */
struct MHOBJect
{
    struct Node mho_Node;           /* embedded Exec node */
    struct List mho_ObjList;        /* embedded List of children objects */
    struct MHOBJect *mho_Parent;    /* pointer to parent object */
    struct MHOBJect *mho_CurNode;   /* pointer to current child object */
    ULONG mho_ID;                  /* numeric ID of object */
    ULONG mho_Status;              /* status of object */
    APTR mho_SysData;              /* message handler data */
    APTR mho_UserData;             /* application data */
    APTR mho_Extens1;              /* *** PRIVATE *** */
    UBYTE mho_Name[1];             /* *** PRIVATE *** */
};
```

The MHOBJect structure contains the following fields:

mho_Node	Embedded Exec list node structure.
mho_ObjList	Embedded Exec list structure for a list of MHOBJect structures. This allows message handler objects to have additional objects attached to them. For example, the IDCMP message handler has window objects, which have gadget objects.
mho_Parent	Pointer to the parent MHOBJect for this object.
mho_CurNode	Pointer to the currently active MHOBJect in the mho_ObjList list.
mho_ID	Numeric ID assigned to this record.
mho_Status	Status of the record, such as enabled or disabled.
mho_SysData	Pointer to data maintained by the message handler.
mho_UserData	This field is provided to the application for its own purposes.

The remaining fields are private to the AppShell.

Message Handler Initialization

At initialization time, each message handler must initialize and return the `MsgHandler` structure. This structure is used by the event processor to manage each message handler.

```
/* message handler structure */
struct MsgHandler
{
    struct MHOBJECT mh_Header;          /* embedded MHOBJECT structure */

    struct MsgPort *mh_Port;            /* message port for handler */
    STRPTR mh_PortName;                /* pointer to the port name, if public */
    ULONG mh_SigBits;                  /* signal bits to watch for */

    /* low level message handler functions */
    WORD mh_NumFuncs;                  /* number of low level msg. handler functions */
    BOOL (**mh_Func)(struct AppInfo *, struct MsgHandler *, struct TagItem *);

    STRPTR *mh_DefText;                /* default text array */
    APTR mh_Catalogue;                 /* *** PRIVATE *** SYSTEM USE ONLY */

    APTR mh_Extens1;                   /* *** PRIVATE *** SYSTEM USE ONLY */
    APTR mh_Extens2;                   /* *** PRIVATE *** SYSTEM USE ONLY */
}
```

The `MsgHandler` structure contains the following components:

<code>mh_Header</code>	Embedded <code>MHOBJECT</code> structure which contains the base information for this message handler.
<code>mh_Port</code>	The message port that this message handler uses.
<code>mh_PortName</code>	A pointer to the constructed public port name for this message handler.
<code>mh_SigBits</code>	The signal bits for the message port.
<code>mh_NumFuncs</code>	Contains the number of low level message handler functions. Four are required: <code>Open</code> , <code>Handle</code> , <code>Close</code> , and <code>Shutdown</code> .
<code>mh_Func</code>	Pointer to an array of low level message handler functions. <code>MH_OPEN</code> , <code>MH_HANDLE</code> , <code>MH_CLOSE</code> , and <code>MH_SHUTDOWN</code> are required.
<code>mh_DefText</code>	Pointer to the default text table for this message handler.

The remainder of the fields are reserved for the AppShell.

The following is an example of a minimal message handler initialization routine.

```
struct MsgHandler *setup_sipcA (struct AppInfo * ai, struct TagItem * tl)
{
    register struct MsgHandler *mh;
    register struct MHOBJect *mho;
    register struct SIPCInfo *md;
    register WORD cntr;
    ULONG msize, hstatus;
    STRPTR pname;
    BOOL exist = TRUE;

    /* calculate the amount of memory that we need */
    msize = sizeof (struct MsgHandler) + sizeof (struct SIPCInfo) +
        (5L * sizeof (ULONG));

    /* allocate instance data */
    if (mh = (struct MsgHandler *) AllocVec (msize, MEMF_CLEAR | MEMF_PUBLIC))
    {
        /* get a pointer to the object */
        mho = &(mh->mh_Header);

        /* initialize the node information */
        mho->mho_Node.ln_Type = MH_HANDLER_T;
        mho->mho_Node.ln_Pri = MH_HANDLER_P;
        mho->mho_Node.ln_Name = "SIPC";

        /* initialize the object list */
        NewList (&(mho->mho_ObjList));

        /* establish the object id */
        mho->mho_ID = APSH_SIPC_ID;

        /* establish the object status */
        mho->mho_Status = (MHS_ENABLED | MHS_CLOSE);

        /* get a pointer to the instance data */
        mho->mho_SysData = md = MEMORY_FOLLOWING (mh);

        /* initialize the message handler functions */
        mh->mh_NumFuncs = 5;
        mh->mh_Func = MEMORY_FOLLOWING (md);
        mh->mh_Func[MH_OPEN] = open_sipc;
        mh->mh_Func[MH_HANDLE] = handle_sipc;
        mh->mh_Func[MH_CLOSE] = close_sipc;
        mh->mh_Func[MH_SHUTDOWN] = shutdown_sipc;
        mh->mh_Func[AH_SENDCMD] = send_sipc_command;

        /* get the activation status */
        hstatus = GetTagData (APSH_Status, NULL, tl);

        /* get the public port name for SIPC */
        strcpy (ai->ai_WorkText, ai->ai_ProgName);
        strcat (ai->ai_WorkText, "_SIPC");
        strupr (ai->ai_WorkText);
        pname = (UBYTE *) GetTagData (APSH_Port, (ULONG)ai->ai_WorkText, tl);
    }
}
```

```

/* allocate room for the port name */
msize = (strlen(pname) + 6L);
mh->mh_PortName = AllocVec (msize, MEMF_CLEAR);
strcpy (mh->mh_PortName, pname);

/* disable multi-tasking for a moment */
Forbid ();

/* get an unique port name */
if (hstatus & P_MULTIPLE)
{
    /* initialize variables */
    exist = TRUE;
    cntr = 1;

    while (exist)
    {
        /* create a name with our base name and a number */
        sprintf (mh->mh_PortName, "%s_%d", pname, cntr);

        /* see if someone has already taken this name */
        if (!FindPort (mh->mh_PortName))
            exist = FALSE;

        cntr++;
    }

    /* create our port */
    if (!(mh->mh_Port = CreatePort (mh->mh_PortName, 0L)))
    {
        /* permit multi-tasking again */
        Permit ();

        /* set up the error return values */
        ai->ai_Pri_Ret = RETURN_FAIL;
        ai->ai_Sec_Ret = APSH_CLDNT_CREATE_PORT;
        ai->ai_TextRtn = PrepText (ai, APSH_MAIN_ID, ai->ai_Sec_Ret,
                                mh->mh_PortName);
    }
    else
        /* permit multi-tasking again */
        Permit ();
}
else
{
    if (FindPort (mh->mh_PortName))
    {
        /* permit multi-tasking again */
        Permit ();

        /* set up the error return values */
        ai->ai_Pri_Ret = RETURN_FAIL;
        ai->ai_Sec_Ret = APSH_PORT_ACTIVE;
        ai->ai_TextRtn = PrepText (ai, APSH_MAIN_ID, ai->ai_Sec_Ret,
                                mh->mh_PortName);
    }
}

```

```

else
{
    /* create our port */
    if (!(mh->mh_Port = CreatePort (mh->mh_PortName, 0L)))
    {
        /* permit multi-tasking again */
        Permit ();

        /* set up the error return values */
        ai->ai_Pri_Ret = RETURN_FAIL;
        ai->ai_Sec_Ret = APSH_CLDNT_CREATE_PORT;
        ai->ai_TextRtn = PrepText (ai, APSH_MAIN_ID, ai->ai_Sec_Ret,
            mh->mh_PortName);
    }
    else
        /* permit multi-tasking again */
        Permit ();
}

if (ai->ai_Pri_Ret == RETURN_OK)
{
    /* set up the signal bits */
    mh->mh_SigBits = (1L << mh->mh_Port->mp_SigBit);

    /* open immediately? */
    if (hstatus & P_ACTIVE)
    {
        if (open_sipc (ai, mh, tl))
            return (mh);
    }
    else
        return (mh);
}

/* make a nice clean failure path */
if (mh->mh_Port)
    DeletePort (mh->mh_Port);
mh->mh_Port = NULL;

/* free the port name */
if (mh->mh_PortName)
    FreeVec ((APTR) mh->mh_PortName);

/* free the memory block */
FreeVec ((APTR) mh);
mh = NULL;
}
else
{
    ai->ai_Pri_Ret = RETURN_FAIL;
    ai->ai_Sec_Ret = APSH_NOT_ENOUGH_MEMORY;
    ai->ai_TextRtn = PrepText (ai, APSH_MAIN_ID, ai->ai_Sec_Ret, NULL);
}

return (mh);
}

```


Message Handler Functions

Message handlers can add functions to the application's function table. The functions must conform to the AppShell function style (outlined in the *Functions* section). The functions must be added to the message handler initialization routine in following manner:

```
/* ID for the Stub function */
#define StubID MYHANDLER_ID+1L

/* function prototype */
VOID StubFunc (struct AppInfo *, STRPTR, struct TagItem *);

/* message handler function table segment */
struct Funcs handler_funcs[] =
{
    {"STUB", StubFunc, StubID,},
    {NULL, NO_FUNCTION,} /* end of array */
};

/* sample message handler initialization routine */
struct MsgHandler * setup_sipcA (struct AppInfo *ai, struct TagItem *tl)
{
    ...

    /* set up the signal bits */
    mh->mh_SigBits = (1L << mh->mh_Port->mp_SigBit);

    /* add the standard functions to the function table */
    AddFuncEntries (ai, handler_funcs);

    ...
}
```

The following is an example of how a standard function can access the message handler's data.

```
/* sample stub function to show how to obtain a message handlers' data */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    register struct MsgHandler * mh;
    register struct MHOject * mho;
    register struct myhInfo * md;

    /* get a pointer to the message handler */
    if (mho=(struct MHOject *)HandlerData(ai,APSH_Handler,"MYH",TAG_DONE))
    {
        /* get a pointer to the message handler data */
        md = mho->mho_SysData;

        ... /* manipulate the message handler data */
    }
}
```

Functions can also call the low-level message handler functions. The following is an example of how a function can access the low-level message handler functions.

```
/* sample stub function to show calling low-level message handler */
/* functions */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    HandlerFunc (ai,
                 APSH_Handler, "IDCMP",
                 APSH_Command, MH_OPEN,
                 APSH_NameTag, "MAIN",
                 TAG_DONE);
}
```

This example would tell the IDCMP (Intuition) message handler to open the window that has the short name "MAIN".

Message Handler Open

Each message handler must have an Open routine that enables the message handler or one of its objects.

Here is an example of a minimal message handler Open routine:

```
/* sample message handler open function */
BOOL open_sipc (struct AppInfo * ai, struct MsgHandler * mh,
               struct TagItem * tl)
{
    register struct MHOBJECT *mho = &(mh->mh_Header);

    mho->mho_Status &= ~MHS_CLOSE;
    mho->mho_Status |= MHS_OPEN;
    return (TRUE);
}
```

Multiple Message Handler Objects

Sometimes it is necessary for a message handler to track several different objects. A custom handler can achieve this by maintaining an Exec list, the MHOBJect field mho_ObjList, of all the available objects. A good example is the IDCMP message handler maintaining multiple windows.

Message Handler Handle

Each message handler must have a Handle routine which manages the messages for its message port. It is in charge of translating these message handler events into commands.

Here is a minimal example of a message handler Handle routine. This example uses the Simple IPC message structure as its form of communication:

```
/* minimal message handler */
BOOL handle_sipc (struct AppInfo * ai, struct MsgHandler * mh,
                  struct TagItem * tl)
{
    register struct MHOBJect *mho = &(mh->mh_Header);
    register struct SIPCMessage *msg;
    WORD FuncID = NO_FUNCTION;
    while (msg = (struct SIPCMessage *) GetMsg (mh->mh_Port))
    {
        /* Get the function number assigned to this message */
        FuncID = (UWORD) msg->sipc_Type;

        if ((FuncID != NO_FUNCTION) && (mho->mho_Status & MHS_OPEN) &&
            (mho->mho_Status & MHS_ENABLED))
        {
            /* Perform the function assigned to this key. */
            PerfFunc (ai, NULL, GetFuncName (ai, (ULONG) FuncID), tl);
        }

        /* reply to the message */
        ReplyMsg ((struct Message *) msg);
    };
    return (TRUE);
}
```

Message Handler Close

Each message handler must have a Close routine that disables the message handler or one of its objects.

Here is an example of a minimal message handler Close routine:

```
BOOL close_sipc (struct AppInfo * ai, struct MsgHandler * mh,
                 struct TagItem * tl)
{
    register struct MHOBJECT *mho = &(mh->mh_Header);

    mho->mho_Status &= ~MHS_OPEN;
    mho->mho_Status |= MHS_CLOSE;
    return (TRUE);
}
```

Message Handler Shut Down

When the ai_Done flag is set to TRUE and the ai_NumCmds field equals zero, the AppShell enters the shut down phase. This involves going through the list of message handlers and calling each handlers' MH_SHUTDOWN routine. This shut down is performed until the list is empty.

Each message handler's shut down routine is in charge of removing the message handler from the message handler list and removing its own resources.

The following is a minimal message handler Shut Down routine.

```
/* minimal message handler shutdown routine */
BOOL shutdown_sipc (struct AppInfo * ai, struct MsgHandler * mh,
                   struct TagItem * tl)
{
    if (mh)
    {
        Remove ((struct Node *) mh);

        /* make sure there is a message port */
        if (mh->mh_Port)
        {
            /* remove the message port */
            DeletePort (mh->mh_Port);
        }

        /* free the message handler data */
        FreeVec ((APTR) mh);
    }
    return (TRUE);
}
```

APPENDIX A - APPSHELL TAGS

Main AppShell Tags

APSH_NumArgs	Number of Shell arguments
APSH_ArgList	Shell arguments
APSH_WBStartup	Workbench arguments
APSH_ControlPort	SIPC Control port for a cloned AppShell
APSH_AppName	Pointer to the application's name
APSH_AppVersion	Pointer to the application's version
APSH_AppCopyright	Pointer to the application's (c) notice
APSH_AppAuthor	Pointer to the application's author
APSH_FuncTable	Function table for application
APSH_DefText	Default text catalogue
APSH_AppInit	Custom application init function ID
APSH_AppExit	Custom application shutdown function ID
APSH_SIG_C	SIG_BREAK_C function ID
APSH_SIG_D	SIG_BREAK_D function ID
APSH_SIG_E	SIG_BREAK_E function ID
APSH_SIG_F	SIG_BREAK_F function ID

Shared System Library handling

APSH_OpenLibraries	Open libraries
APSH_LibNameTag	Library name tag
APSH_LibName	Library name
APSH_LibVersion	Library version
APSH_LibStatus	Required/optional
APSH_LibBase	Library base
APSH_ARexxSys	rexsyslib.library
APSH_ARexxSup	rexsupport.library
APSH_AS_L	asl.library
APSH_Commodities	commodities.library
APSH_DiskFont	diskfont.library
APSH_DOS	dos.library
APSH_GadTools	gadtools.library
APSH_Gfx	graphics.library
APSH_Icon	icon.library
APSH_Intuition	intuition.library
APSH_Layers	layers.library
APSH_IFF	iffparse.library
APSH_Translate	translator.library
APSH_Utility	utility.library
APSH_Workbench	workbench.library

Message Handler Routines

APSH_AddHandler	Add a message handler to application
APSH_Setup	Setup function
APSH_Status	Active, inactive, multiple, etc...
APSH_Rating	Optional/required, etc...
APSH_Port	Name of the message port
APSH_Handler	Handler ID
APSH_CmdData	Command data
APSH_CmdDataLength	Length of command data
APSH_CmdID	Command ID (function)
APSH_CmdString	Command string
APSH_CmdTagList	Command tag list
APSH_Command	Handler command
APSH_NameTag	Name Tag for object

ARexx Information

APSH_Extens	ARexx macro name extension
APSH_ARexxError	ARexx command ERROR function ID
APSH_ARexxOK	ARexx command OK function ID

Command Shell

APSH_CloseMsg	Closing message
APSH_CMDWindow	Command window spec
APSH_Prompt	Command window prompt

Window Information

APSH_WindowEnv	Window Environment
APSH_TextAttr	Text Attributes
APSH_NewScreen	NewScreen structure
APSH_NewScreenTags	Tags for new screen
APSH_Palette	Color Palette
APSH_NewWindow	NewWindow structure
APSH_NewWindowTags	Tags for new window
APSH_HotKeys	HotKey command array
APSH_Menu	Intuition-style Menu array
APSH_Gadgets	Intuition-style Gadget array
APSH_GTMenu	GadTools-style Menu array
APSH_GTGadgets	GadTools-style NewGadget array
APSH_GTFlags	flags for GadTools objects
APSH_Objects	Object array
APSH_ObjDown	Gadget downpress function ID
APSH_ObjHold	Gadget hold function ID
APSH_ObjRelease	Gadget release function ID
APSH_ObjDb1Click	Gadget double-click function ID
APSH_ObjAbort	Gadget abort function ID
APSH_ObjAltHit	Gadget ALT hit function ID
APSH_ObjShiftHit	Gadget SHIFT hit function ID
APSH_ObjData	Gadget image or data
APSH_ObjInner	Inner rectangle
APSH_ObjPointer	Pointer name prefix
APSH_DefWinFlags	Default window flags

IDCMP Messages

APSH_SizeVerify	SIZEVERIFY function ID
APSH_NewSize	NEWSIZE function ID
APSH_RefreshWindow	REFRESHWINDOW function ID
APSH_MouseButtons	MOUSEBUTTONS function ID
APSH_ReqSet	REQSET function ID
APSH_CloseWindow	CLOSEWINDOW function ID
APSH_ReqVerify	REQVERIFY function ID
APSH_ReqClear	REQCLEAR function ID
APSH_MenuVerify	MENUVERIFY function ID
APSH_DiskInserted	DISKINSERTED function ID
APSH_DiskRemoved	DISKREMOVED function ID
APSH_ActiveWindow	ACTIVEWINDOW function ID
APSH_InactiveWindow	INACTIVEWINDOW function ID

Real or Simulated IntuiMessage fields

APSH_MsgClass	Message class
APSH_MsgCode	Message code
APSH_MsgQualifier	Message qualifier
APSH_MsgIAddress	Item address
APSH_MsgMouseX	Mouse X coordinate
APSH_MsgMouseY	Mouse Y coordinate
APSH_MsgSeconds	Seconds
APSH_MsgMicros	Micros
APSH_MsgWindow	Window for event

SIPC Message

APSH_SIPCData	Pointer the data passed by a SIPC message
APSH_SIPCDataLength	Length of the SIPC data

Standard Tool Information

APSH_Tool	Name of tool
APSH_ToolAddr	Address of tool
APSH_ToolData	Data for tool
APSH_ToolStack	Stack requirements of tool
APSH_ToolPri	Tool process priority

APPENDIX B - DEFAULT APPSHELL TEXT TABLE

Text ID	Default Text
APSH_NOT_AN_ICON	%s is not an icon.
APSH_NOT_AVAILABLE	%s is not available
APSH_PORT_ACTIVE	%s port already active
APSH_PORT_X_ACTIVE	port, %s, already active
APSH_NOT_AN_IFF	%s is not an IFF file
APSH_NOT_AN_IFF_X	%1\$s is not an IFF %2\$s file
APSH_CLOSE_ALL_WINDOWS	Close all windows
APSH_CMDSHELL_PROMPT	Cmd>
APSH_CLDNT_CREATE_X	Could not create %s
APSH_CLDNT_CREATE_PORT	Could not create port, %s
APSH_CLDNT_CREATE_OBJ	Could not create object
APSH_CLDNT_CREATE_OBJ_X	Could not create object, %s
APSH_CLDNT_CREATE_FILE	Could not create file
APSH_CLDNT_CREATE_FILE_X	Could not create file, %s
APSH_CLDNT_INIT_X	Could not initialize %s
APSH_CLDNT_INIT_MSGH	Could not initialize %s message handler
APSH_CLDNT_LOCK	Could not lock %s
APSH_CLDNT_LOCK_DIR	Could not lock directory
APSH_CLDNT_LOCK_DIR_X	Could not lock directory, %s
APSH_CLDNT_LOCK_PUB	Could not lock public screen
APSH_CLDNT_LOCK_PUB_X	Could not lock public screen, %s
APSH_CLDNT_OBTAIN	Could not obtain %s
APSH_CLDNT_OPEN	Could not open %s
APSH_CLDNT_OPEN_FILE	Could not open file
APSH_CLDNT_OPEN_FILE_X	Could not open file, %s
APSH_CLDNT_OPEN_FONT_X	Could not open font, %s
APSH_CLDNT_OPEN_MACRO	Could not open macro file, %s
APSH_CLDNT_OPEN_PREF	Could not open preference file, %s
APSH_CLDNT_OPEN_SCREEN	Could not open screen
APSH_CLDNT_OPEN_WINDOW	Could not open window
APSH_SETUP_TIMER	Could not set up timer event
APSH_SETUP_HOTKEYS	Could not set up HotKeys
APSH_START_PROCESS	Could not start process
APSH_START_TOOL	Could not start tool
APSH_START_TOOL_X	Could not start tool, %s
APSH_WRITE_FILE	Could not write to file
APSH_WRITE_FILE_X	Could not write to file, %s
APSH_WRITE_MACRO	Could not write to macro file
APSH_CMDSHELL_WIN	CON:0/150/600/50/Command Shell/CLOSE
APSH_NO_NAMETAG_WIN	No name given for window
APSH_NO_PORT	No port name specified
APSH_NOT_ENOUGH_MEMORY	Not enough memory
APSH_WAITING_FOR_MACRO	Waiting for macro return
APSH_DISABLED	%s is disabled



AppShell AutoDoc

TABLE OF CONTENTS

Link Library Functions

appshell.lib/HandleApp
appshell.lib/NotifyUser

Shared System Library Functions

appshell.library/APSHGetGadgetInfo
appshell.library/BuildParseLine
appshell.library/FindType
appshell.library/FreeParseLine
appshell.library/GetText
appshell.library/HandleAppAsync
appshell.library/HandlerData
appshell.library/HandlerFunc
appshell.library/MatchValue
appshell.library/ParseLine
appshell.library/PerfFunc
appshell.library/PrepText
appshell.library/QStrCmpI
appshell.library/RemoveMsgPort

Standard Function Table Commands

appshell.library/ACTIVATE
appshell.library/ALIAS
appshell.library/CMDSHELL
appshell.library/DISABLE
appshell.library/ENABLE
appshell.library/HOTKEY
appshell.library/RX
appshell.library/TOBACK
appshell.library/TOFRONT
appshell.library/VERSION
appshell.library/WHY
appshell.library/WINDOW

NAME

HandleApp - startup function for AppShell application

SYNOPSIS

```
results = HandleApp (argc, argv, wbm, attrs)
```

```
BOOL results;  
int argc;  
char **argv;  
struct WBStartup *wbm;  
struct TagItem *attrs;
```

FUNCTION

This function is the application entry point to the AppShell.

EXAMPLE

```
/* ... application description tags ... */  
  
extern struct WBStartup *WBenchMsg;  
  
void main(int argc, char **argv)  
{  
    HandleApp(argc, argv, WBenchMsg, tags);  
}
```

INPUTS

argc - Number of Shell arguments being passed.
argv - Pointer to the Shell argument list
wbm - Pointer to the Workbench startup message.
attrs - Pointer to the application's user interface description.

RESULT

TRUE - Application was able to initialize and run.

FALSE - Application failed. AppShell has already informed user of what and where the failure was.

SEE ALSO

HandleAppAsync()

NAME

NotifyUser - Display a text message to the user.

SYNOPSIS

NotifyUser (ai, msg, tl)

```
struct AppInfo * ai;  
STRPTR msg;  
struct TagItem * tl;
```

FUNCTION

This function will display a text message to the user. Currently uses EasyRequest (AutoRequest if under 1.3) to display messages.

INPUTS

ai - Optional pointer to the AppInfo structure for this application.

msg - Pointer to message to display. If this field is NULL, then the text will come from the ai->ai_TextRtn field.

tl - Optional pointer to an array of TagItems.

NAME

APSHGetGadgetInfo - Obtain a pointer to a gadget and its window.

SYNOPSIS

```
success = APSHGetGadgetInfo (ai, winname, gadname, winptr, gadptr)
```

```
BOOL success;
struct AppInfo *ai;
STRPTR winname, gadname;
ULONG *winptr, *gadptr;
```

FUNCTION

Obtain a pointer to a gadget and the window that it belongs in.

This function is implemented by the IDCMP message handler.

EXAMPLE

```
/* Sample function showing how to use APSHGetGadgetInfo */
VOID StubFunc (struct AppInfo * ai, STRPTR cmd, struct TagItem * tl)
{
    struct Gadget *gad;
    struct Window *win;

    /* Get a pointer to the main window, named MAIN, and the Scrolling
     * List gadget, which is named LIST.
     */
    if (APSHGetGadgetInfo (ai, "MAIN", "LIST",
                          (ULONG *)&win, (ULONG *)&gad))
    {
        /* ... do something with the fields ... */
    }
}
```

INPUTS

ai	- Pointer to the AppInfo structure
winname	- Pointer to the name of the window that the gadget is supposed to be located in.
gadname	- Pointer to the name of the gadget.
winptr	- Address of the variable to place the window pointer in.
gadptr	- Address of the variable to place the gadget pointer in.

RETURN

success - TRUE indicates that the system was able to locate both the named window and gadget.

FALSE indicates that the system was unable to locate either the window or the gadget.

NAME

BuildParseLine - Non-destructive string parser

SYNOPSIS

```
handle = BuildParseLine (line, argc, argv);
```

```
STRPTR handle;
STRPTR line;
ULONG *argc;
STRPTR argv[MAXARGS];
```

FUNCTION

This function is used to parse a string that may end up being passed to another function. It does not modify the passed string. Requires a corresponding call to FreeParseLine, when the application's function is done with the parsed line.

INPUTS

```
line   - Pointer to the string to parse.
argc   - Pointer to variable to hold the number of arguments.
argv   - Pointer to an array to hold the arguments. The array
         must contain MAXARGS entries.
```

RETURN

handle - Pointer to a temporary work area which must be passed back to FreeParseLine when done with the parsed array.

EXAMPLE

```
/* sample function showing how to use BuildParseLine & FreeParseLine */
function()
{
    STRPTR argv[MAXARGS], handle = NULL;
    ULONG argc;

    /* Parse the command line */
    handle = BuildParseLine ("ACTIVATE ME", &argc, argv);

    /* Do something with the parsed command line. The strings
     * must be copied before calling FreeParseLine if they are
     * going to be used later.
     */
    printf("%ld words, first is %s\n", argc, argv[0]);

    /* free the BuildParseLine resources */
    FreeParseLine(handle);
}
```

SEE ALSO

FreeParseLine(), ParseLine()

NAME

FindType - Find the value of a variable.

SYNOPSIS

```
value = FindType(argv, name, defvalue)
```

```
STRPTR value;
STRPTR argv[MAXARGS];
STRPTR name, defvalue;
```

FUNCTION

This function searches a parsed text array for a given entry and returns a pointer to the value bound to that entry. If the entry is not found, then a pointer to defvalue is returned.

EXAMPLE

```
/* sample fragment showing how to use FindType */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    STRPTR name;
    STRPTR argv[MAXARG], clone=NULL;
    ULONG argc;

    /* make sure we have a command line */
    if (cmd)
    {
        /* parse the command line */
        clone = BuildParseLine (cmd, &argc, argv);

        /* Get the file name. If there isn't a FILE keyword, then
         * use "config". If the return value is to be used outside
         * of this function, then it must be copied.
         */
        name = FindType (argv, "FILE", "config");
    }

    /* free the BuildParseLine */
    FreeParseLine (clone);
}
```

INPUTS

```
argv    - Pointer to the preparsed text array.
name    - Entry to search for.
defvalue - Value to return if name isn't found.
```

RESULTS

```
value    - A pointer to the string that is the
           value bound to name or defvalue if name isn't found.
```

SEE ALSO

```
MatchValue(), BuildParseLine(), FreeParseLine(), ParseLine(),
icon.library/FindToolType
```

NAME

FreeParseLine - Free the BuildParseLine temporary work area

SYNOPSIS

FreeParseLine(handle)

STRPTR handle;

FUNCTION

Free the temporary work space used by BuildParseLine. A NULL is a valid argument.

INPUTS

handle - Pointer to the return value from BuildParseLine.

SEE ALSO

BuildParseLine(), ParseLine()

NAME

GetText - Obtain a pointer to a permanent read-only text message.

SYNOPSIS

```
text = GetText(ai, base, id, def)
```

```
STRPTR text;  
struct AppInfo *ai;  
ULONG base, id;  
STRPTR def;
```

FUNCTION

Used to obtain a pointer to a permanent text message. The text return value must not be modified.

The main use for this function is for setting up text labels.

EXAMPLE

```
/* sample function to show how to use GetText */  
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)  
{  
    STRPTR label;  
  
    /* get the current text to use for Okay */  
    label = GetText (ai, APSH_MAIN_ID, APSH_OKAY_TXT, NULL);  
}
```

INPUTS

ai - Pointer to the AppInfo structure for this application.
base - Text table to use.

APSH_USER_ID for the application text table.
APSH_MAIN_ID for the AppShell text table.
or may provide a custom message handler base ID.

id - Text Table entry id.
def - Must be NULL.

SEE ALSO

PrepText()

NAME

HandleAppAsync - startup function for an asynchronous AppShell application.

SYNOPSIS

```
results = HandleAppAsync (attrs, sipc)
```

```
BOOL results;
struct TagItem *attrs;
struct MsgPort *sipc;
```

FUNCTION

This function is the application entry point for an asynchronous AppShell application. This function should be called via the Tool Message Handler and allows an AppShell application to have multiple projects each with their own process.

EXAMPLE

```
/* This example shows how an application can clone itself to handle
 * multiple projects.
 *
 * APSH_Tool    the name to give to the cloned process.
 * APSH_ToolAddr the function to run asynchronously.
 * APSH_ToolData the user interface tag description.
 */
VOID CloneFunc(struct AppInfo *ai, STRPTR args, struct TagItem *tl)
{
    HandlerFunc (ai,
        APSH_Handler,  "TOOL",
        APSH_Command,  MH_OPEN,
        APSH_Tool,     "AppShell_Clone",
        APSH_ToolAddr,  HandleAppAsync,
        APSH_ToolData,  Cloned_App,
        TAG_DONE);
}
```

INPUTS

attrs - Pointer to the application's user interface description.

sipc - SIPC message port by which the master application can control the cloned application.

RESULT

TRUE - Application was able to initialize and run.

FALSE - Application failed. AppShell has already informed user of what and where the failure was.

SEE ALSO

HandleApp()

NAME

HandlerData - Obtain a pointer to a message handlers' instance data

SYNOPSIS

HandlerData (struct AppInfo *ai, ULONG tags, ...)

```
struct AppInfo *ai;
ULONG tags, ...
```

OR

HandlerDataA (struct AppInfo *ai, struct TagItem *tl)

```
struct AppInfo *ai;
struct TagItem *tl;
```

FUNCTION

Used to obtain a pointer to a message handlers' instance data. HandlerData is the stack-based variable argument interface, while HandlerDataA is the TagItem array interface.

EXAMPLE

```
/* sample stub function to show how to obtain a message handlers'
 * data
 */
VOID StubFunc(struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    struct MsgHandler *mh;
    struct MHObject *mho;
    struct myhInfo *md;

    /* get a pointer to the message handler data */
    if (mho = (struct MHObject *)
        HandlerData(ai, APSH_Handler, "MYH", TAG_DONE))
    {
        /* get a pointer to the instance data */
        md = mho->mho_SysData;
    }
}
```

INPUTS

ai - pointer to the AppInfo structure for this application.
tags - stack based TagItems.

SEE ALSO

HandlerFunc()

NAME

HandlerFunc - entry point for a low-level message handler function.

SYNOPSIS

HandlerFunc (struct AppInfo *ai, ULONG tags, ...)

```
struct AppInfo *ai;
ULONG tags, ...
```

OR

HandlerFuncA (struct AppInfo *ai, struct TagItem *tl)

```
struct AppInfo *ai;
struct TagItem *tl;
```

FUNCTION

Provides an entry point for the low-level message handler functions. HandlerFunc is the stack-based variable argument interface, while HandlerFuncA is the TagItem array interface.

EXAMPLE

```
/* sample stub function to show how to call a low-level message
 * handler function.
 *
 * This example tells the IDCMP message handler to open the window
 * (and it's environment) named MAIN.
 */
VOID StubFunc(struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    HandlerFunc(ai,
        APSH_Handler, "IDCMP",
        APSH_Command, MH_OPEN,
        APSH_NameTag, "MAIN",
        TAG_DONE);
}
```

INPUTS

ai - pointer to the AppInfo structure for this application.
tags - stack based TagItems.

SEE ALSO

HandlerData()

NAME

MatchValue - Check a text argument for a particular flag.

SYNOPSIS

```
value = MatchValue(entry, value)
```

```
BOOL value;
STRPTR entry, value;
```

FUNCTION

This function searches to see if a particular text flag is set in a text entry.

EXAMPLE

```
/* sample fragment showing how to use MatchValue */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    STRPTR flagstr;
    ULONG flags = NULL;
    STRPTR argv[MAXARG], clone=NULL;
    ULONG argc;

    /* make sure we have a command line */
    if (cmd)
    {
        /* parse the command line */
        clone = BuildParseLine (cmd, &argc, argv);

        /* get the flag entry from the argument list */
        if (flagstr = FindType (argv, "FLAGS", NULL))
        {
            /* see if the CLOSE flag is present */
            if (MatchValue (flagstr, "CLOSE"))
                flags |= CLOSEWINDOW;

            /* see if the SIZE flag is present */
            if (MatchValue (flagstr, "SIZE"))
                flags |= NEWSIZE;
        }
    }

    /* free the BuildParseLine */
    FreeParseLine (clone);
}
```

INPUTS

```
entry    - Entry to search in.
value    - Value to search for.
```

RESULTS

```
value    - TRUE if the value was in the entry, otherwise returns
            FALSE.SEE ALSO FindType(), BuildParseLine(), FreeParseLine(),
            ParseLine(), icon.library/MatchToolValue
```

NAME

ParseLine - Destructive string parser

SYNOPSIS

```
argc = ParseLine (line, argv);
```

```
ULONG argc;  
STRPTR line;  
STRPTR argv[MAXARGS];
```

FUNCTION

String parser. Inserts '0' after each word in the passed text line.

INPUTS

line - Pointer to the string to parse.
argv - Pointer to an array to hold the arguments. The array must contain MAXARGS entries.

RETURN

argc - Number of arguments returned in argv

EXAMPLE

```
/* sample function showing how to use ParseLine */  
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)  
{  
    UBYTE text[] = "ACTIVATE ME";  
    STRPTR argv[MAXARGS];  
    ULONG argc;  
  
    /* parse the command line */  
    argc = ParseLine (text, argv);  
  
    /* simple display of some of the parse variables */  
    printf("%ld words, first is %s\n", argc, argv[0]);  
}
```

SEE ALSO

BuildParseLine(), FreeParseLine()

NAME

PerfFunc - The entry point to all commands in the function table.

SYNOPSIS

PerfFunc (ai, fid, cmdline, attrs)

```
struct AppInfo *ai;
ULONG fid;
STRPTR cmdline;
struct TagItem *attrs;
```

FUNCTION

This is the main and only entry point for all commands in the function table. In order to respect state, such as whether the function is enabled or disabled, function table commands should never be called directly.

If the function isn't in the function table and the ARexx message handler has been initialized, then the command is passed to ARexx.

EXAMPLE

```
/* how to call a function using its ID */
PerfFunc (ai, QuitID, NULL, NULL);

/* how to call a function using its name */
PerfFunc (ai, NULL, "QUIT", NULL);
```

INPUTS

ai	- Pointer to the AppInfo structure
fid	- ID of function to perform.
cmdline	- Pointer to the text command line to use to trigger the command.
attrs	- Pointer to the TagItem array to use to triggered the command.

NAME

PrepText - Obtain a pointer to a temporary modifiable text message.

SYNOPSIS

```
text = PrepText(ai, base, id, args, ...)
```

```
STRPTR text;
struct AppInfo *ai;
ULONG base, id;
APTR args;
```

FUNCTION

Build a temporary text message using a text table entry and the passed arguments. The text table entry must contain valid RawDoFmt formatting commands.

The text pointer is only valid until the next call to PrepText. There is one PrepText buffer per AppInfo, so each cloned AppShell has its own work buffer. The string must be copied to a more permanent storage place if you wish to keep it for a longer period.

The main use for this function is for formatting of temporary error messages.

EXAMPLE

```
/* set up error return values */
ai->ai_Pri_Ret = RETURN_FAIL;
ai->ai_Sec_Ret = APSH_PORT_ACTIVE;

/* "%s port already active", pname */
ai->ai_TextRtn = PrepText(ai, APSH_MAIN_ID, ai->ai_Sec_Ret, pname);
```

INPUTS

ai - Pointer to the AppInfo structure for this application.
base - Text table to use.

APSH_USER_ID for the application text table.
APSH_MAIN_ID for the AppShell text table.
or may provide a custom message handler base ID.

id - Text Table entry id.
args - Variables to be sprintf'ed into the text entry.

SEE ALSO

GetText()

NAME

QStrCmpI - Quick case insensitive string comparision.

SYNOPSIS

```
success = QStrCmpI (str1, str2);
```

BOOL value;

STRPTR str1, str2;

FUNCTION

This function performs a quick, case insensitive, string comparision. Stops as soon as it determines that the strings are not the same.

EXAMPLE

```
/* sample code fragment showing how to use QStrCmpI */
VOID StubFunc (struct AppInfo *ai, STRPTR cmd, struct TagItem *tl)
{
    STRPTR name;
    STRPTR argv[MAXARG], clone=NULL;
    ULONG argc;

    /* make sure we have a command line */
    if (cmd)
    {
        /* parse the command line */
        clone = BuildParseLine (cmd, &argc, argv);

        /* make sure we have some arguments */
        if (argc >= 2L)
        {
            /* check to see if the first argument is CLOSE.
             * Note that FindType(argv, "CLOSE", NULL) could also be
             * used in this example.
             */
            if (QStrCmpI (argv[1], "CLOSE"))
            {
                /* do something because of CLOSE */
            }
        }

        /* free the BuildParseLine */
        FreeParseLine (clone);
    }
}
```

INPUTS

str1 - Pointer to the first string.
str2 - Pointer to the second string.

RESULTS

value - TRUE if the strings are the same, otherwise returns FALSE.

NAME

RemoveMsgPort - Safely remove a message port.

SYNOPSIS

RemoveMsgPort (mp);

struct MsgPort *mp;

FUNCTION

This function will remove and reply to all messages that are outstanding on a message port before removing the port itself.

NULL is a valid argument.

INPUTS

mp - A pointer to the message port to delete.

NAME

ACTIVATE - Activate an AppShell window.

SYNOPSIS

ActivateID Function ID
ActivateFunc Function prototype

FUNCTION

Provides a mechanism to activate an AppShell window. Activation involves bring the screen to front, opening the window if it is hidden, bringing the window to front and unzooming the window if it is zoomed.

As a string command line:

ACTIVATE [name]

where [name] is a valid window name, defaults to MAIN.

As a TagItem attribute list:

APSH_NameTag, <name>
where <name> is a valid window name.

This function is implemented by the IDCMP message handler.

SEE ALSO

TOBACK, TOFRONT, WINDOW

NAME

ALIAS - Used to build new commands from existing commands.

SYNOPSIS

AliasID	Function ID
AliasFunc	Function prototype

FUNCTION

This command is used to build new commands from existing commands in the function table.

As a string command line:

ALIAS <new> <existing> [parameters]

where <new> is the new command to add to the function table.

where <existing> is an existing command in the function table to assign to the new command.

where [parameters] are optional parameters for <existing> to assign to <new>.

EXAMPLE

The following command line would assign the command OPEN to the new command name README, and would use READ.ME as the parameter to pass.

ALIAS README OPEN READ.ME

BUGS

No tags are implemented.

NAME

CMDSHELL - Open/Close the application command shell.

SYNOPSIS

CMDShellID Function ID
CMDShellFunc Function prototype

FUNCTION

Opens a console window whereby the user can interact directly with the application at a command level. Allows the user quick access to functions or macros that they may use so infrequently that they don't wish to bind them to a key, menu or button.

As a string command line:

CMDSHELL [command]
where [command] is a valid command, defaults to OPEN.

OPEN Open the command shell.
CLOSE Close the command shell.

As a TagItem attribute list:

APSH_Command, <command>
where <command> is a valid command, defaults to MH_OPEN.

MH_OPEN Open the command shell.
MH_CLOSE Close the command shell.

This function is implemented by the DOS message handler.

NAME

DISABLE - Disable a function and its interfaces.

SYNOPSIS

DisableID Function ID
DisableFunc Function prototype

FUNCTION

This function allows an application or user to disable a function.

If the function is attached to a gadget or menu item, then that item is also disabled.

As a string command line:

DISABLE [name]

where [name] is a valid function name.

As a TagItem attribute list:

APSH_NameTag, <name>

where <name> is a valid function name.

SEE ALSO

ENABLE

NAME

ENABLE - Enable a function and its interfaces.

SYNOPSIS

EnableID Function ID
EnableFunc Function prototype

FUNCTION

This function allows an application or user to enable a function.

If the function is attached to a gadget or menu item, then that item is also enabled.

As a string command line:

ENABLE [name]

where [name] is a valid function name.

As a TagItem attribute list:

APSH_NameTag, <name>

where <name> is a valid function name.

SEE ALSO

DISABLE

NAME

HOTKEY - Bind a keyboard command to a function.

SYNOPSIS

HotKeyID Function ID
HotKeyFunc Function prototype

FUNCTION

Provides a mechanism to bind a function to a keystroke or sequence of keystrokes.

As a string command line:

HOTKEY <key> <function name>

where <key> is the keystroke.

where <function name> is a valid function name.

As a TagItem attribute list:

APSH_CmdData, <key>
where <key> is the keystroke.

APSH_NameTag, <function name>
where <function name> is a valid function name.

This function is implemented by the IDCMP message handler.

BUGS

Currently doesn't support anything but single, shifted and unshifted, alphabetic characters [when called from the command line].

NAME

RX - Pass a command through ARexx.

SYNOPSIS

RXID Function ID
RXFunc Function prototype

FUNCTION

Allows access to ARexx functions that may have otherwise been rerouted by the application. Should only be used if there are name conflicts between an application function and an ARexx function.

As a string command line:

RX [command]

where [command] is a command line to pass to ARexx.

As a TagItem attribute list:

APSH_CmdString, <command>

where <command> is a command line to pass to ARexx.

This function is implemented by the ARexx message handler.

BUGS

Not currently implemented.

NAME

TOBACK - Send an AppShell window to the back.

SYNOPSIS

ToBackID Function ID
ToBackFunc Function prototype

FUNCTION

Provides a mechanism to send an AppShell window to the back of the other windows on the same screen.

As a string command line:

TOBACK [name]

where [name] is a valid window name, defaults to MAIN.

As a TagItem attribute list:

APSH_NameTag, <name>
where <name> is a valid window name.

This function is implemented by the IDCMP message handler.

SEE ALSO

ACTIVATE, TOFRONT, WINDOW

NAME

TOFRONT - Bring an AppShell window to the front.

SYNOPSIS

ToFrontID Function ID
ToFrontFunc Function prototype

FUNCTION

Provides a mechanism to bring an AppShell window in front of the other windows on the same screen.

As a string command line:

TOFRONT [name]

where [name] is a valid window name, defaults to MAIN.

As a TagItem attribute list:

APSH_NameTag, <name>
where <name> is a valid window name.

This function is implemented by the IDCMP message handler.

SEE ALSO

ACTIVATE, TOBACK, WINDOW

NAME

VERSION - Uses NotifyUser() to show the current version.

SYNOPSIS

VersionID Function ID
VersionFunc Function prototype

FUNCTION

Uses NotifyUser() to show the current version of the application or the AppShell.

As a string command line:

VERSION [APPSHELL]

[APPSHELL] To display the AppShell version. Defaults
to showing the application version.

BUGS

No tags are implemented.

NAME

WHY - Return information on the last error.

SYNOPSIS

WhyID Function ID
WhyFunc Function prototype

FUNCTION

For scripting purpose, the primary return value of a command should return an error level. This limits the error return values to small ranges (to be consistent with DOS). Therefore, if commands use the secondary return field to contain information on the actual error, then this function will allow scripts to obtain information on what the last error actually was.

This command takes no parameters.

This function is implemented by the ARexx message handler.

BUGS

Not currently implemented.

NAME

WINDOW - Open/Close an AppShell window

SYNOPSIS

WindowID Function ID
WindowFuncFunction prototype

FUNCTION

Provides a mechanism to open or close an AppShell window.

As a string command line:

WINDOW [action] [name]

where [action] is an optional command, defaults to OPEN.

OPEN To open the window.

CLOSE To close the window.

where [name] is a valid window name, defaults to MAIN.

As a TagItem attribute list:

APSH_NameTag, <name>

where <name> is a valid window name.

APSH_Command, <cmd>

where <cmd> is a valid command.

MH_OPEN To open the window.

MH_CLOSE To close the window.

This function is implemented by the IDCMP message handler.

SEE ALSO

ACTIVATE, TOBACK, TOFRONT



Intuition V2.0 Documentation Update

by Jim Mackraz

This document provides an overview of Intuition V2.0, that is, intuition.library version 36. It supercedes notes provided at earlier Developer's Conferences and information provided with alpha and beta releases.

We only provide an quick overview of the various topics here. Detailed specifications for new features are provided by the accompanying "autodocs" for intuition.library, and by the Intuition include files.

General Information

TagItem Structures and Tag Lists. Throughout the operating system, a new parameter and storage mechanism has been introduced. Defined in the include file *utility/tagitem.h*, and supported by routines in *utility.library*, arrays of tagged data items are used in several places in Intuition to specify attribute/value pairs to functions such as *OpenWindow()*, *OpenScreen()* and new functions such as *SetGadgetAttrs()*.

In C, a tagged data item is a *TagItem* structure, defined below. The field *ti_Data* often contains 32-bit data which is cast to a *ULONG*.

```
struct TagItem
{
    ULONG    ti_Tag;        /* identifies the type of this item */
    ULONG    ti_Data;       /* type-specific data, can be a pointer */
};
```

A "tag list" is an array or chain of arrays of *TagItem* structures, specifying a set of attribute/value pairs.

Unless documentation for a particular use of tag lists states otherwise, you should avoid duplicate *ti_Tag* entries in a tag list and not make any assumptions about the order in which tag items in a list are processed.

Within this array, different data items are identified by the value in *ti_Tag*. Items specific to an application have a *ti_Tag* value which has the *TAG_USER* bit set. System data items

which can be used by any application have a `ti_Tag` value with `TAG_USER` bit clear. These include:

- `TAG_IGNORE` - a no-op. The data item is ignored.
- `TAG_MORE` - the `ti_Data` points to another array of tag items, to support "chaining" of `TagItem` arrays.
- `TAG_DONE` - terminates the tag item array (or chain).
- `TAG_SKIP` - ignore the current tag item, and skip succeeding array elements, `ti_Data` in number.

Note that user tags need only be unique within the particular context of their use. For example, the attribute tags defined for `OpenWindow()` overlap numerically with some tag values used by `OpenScreen()`, but the same numeric value has different meaning in the different contexts.

You can use tag lists for your own functions. There are numerous support functions in *utility.library*, including `FindTagItem()`, `GetTagData()`, and `NextTagItem()`. The last is for iterating all the items in a tag list and is at the root of almost all the others.

There is also a particularly handy function, `PackBoolTags()`, which collapses the Boolean attributes in a tag list into corresponding bits in a flag word. See the autodocs for *utility.library* for the details. `TagItems` are no longer part of Intuition.

Passing Tag Lists to System Functions – varargs. You can provide `TagItems` to functions in several ways. Functions `OpenWindow()` and `OpenScreen()` support an extension to the structures `NewWindow` and `NewScreen`, respectively, which is a pointer to a tag list of window or screen attribute specifications.

There are also two new functions `OpenWindowTagList()` and `OpenScreenTagList()` to which you pass a `NewWindow` and a tag list, or a `NewScreen` and a tag list.

There are also "varargs" versions of these routines implemented in *amiga.lib*, `OpenWindowTags()` and `OpenScreenTags()`. "Varargs" is a C term used to indicate a function which takes a variable number of arguments, such as "printf."

As an example, here is a complete code fragment to open a screen of "text overscan" dimensions ("MoreRows" dimensions):

```
screen = OpenScreen( NULL, /* don't bother with a NewScreen */
    SA_Overscan, OSCAN_TEXT,
    SA_Title, "My Screen",
    TAG_DONE );
```

This example illustrates the convenience of a varargs interface to `OpenScreen()`, and also that

Intuition supports a lot of default values for screen parameters.

Other parameters besides tag lists can be provided to Intuition with a varargs interface. The function `EasyRequest()` takes printf-style arguments to fill in the “blanks” in printf-style format strings. There is always a corresponding non-varargs interface underlying all varargs system calls – in this case `EasyRequestArgs()`.

The name variations distinguishing two similar functions differing only by a varargs interface are not consistently applied, unfortunately.

Function Callback Hooks. Another standard piece of technology used in various places by Intuition is the standard application-callback interface implemented by the Hook data structure. These are defined in *utility/hooks.h*, and specify a register-based parameter convention. Examples of calling through a hook and being called via a hook are provided in the Intuition examples.

There were already mechanisms such as interrupts and input handlers where the system calls user code but the interface to high-level languages (HLLs) is rarely standardized. The goals of Hooks include:

- ☐ Standard calling conventions are respected by using a register parameter convention. Any language which can implement an interrupt handler should be able to support Hooks.
- ☐ A single assembly language interface routine can be used to interface to several HLL hook routines.
- ☐ A standard method of specifying HLL entry points is used so that portability of HLL implementations between compilers is possible.
- ☐ Future enhancement of the technique is possible.

IntuitionBase and Private Data. As promised, the non-public portion of the IntuitionBase data structure was completely revised for this version of Intuition. There is no public information on what it has turned into, nor how it is used.

Likewise, new additional data for Screen and Window structures has been added without publication in public include files. If there is some chore your application cannot accomplish without accessing private data, you are invited to request proper programmatic access to it in the next release.

The New Look, DrawInfo and Pen Specifications. As will be discussed below in a little more detail, the new look of Intuition displays is centered around a per-screen data structure called a DrawInfo. This structure encapsulates all information that Intuition uses for rendering the new look, including specification of which pen numbers are used for “shadow,” “shine” and “text.” Screen resolution values are also contained as is other useful information.

See the definition of struct DrawInfo in *intuition/screens.h*.

Stack Swapping. With the introduction of “New Look” system requesters, the function AutoRequest() started using considerably more processor stack than it had in previous versions of Intuition. This caused some severe and frequent problems, so code was installed to switch stack (unconditionally) when AutoRequest() and EasyRequest() (and their ilk) are called.

It turned out that both DOS and the File System had identical code to guarantee safe stacks before calling these routines, so the system enjoyed a net shrinkage in ROM space by centralizing this code in Intuition.

Changes to Windows

The most important new change is the addition of a tag list specification to OpenWindow(), via either the structure ExtNewWindow or the new functions OpenWindowTagList() and OpenWindowTags(). See *intuition/intuition.h* for a complete list of window attribute tags, and the autodocs for detailed explanation of their uses.

Windows borders are rendered quite differently in V2.0, as you’ve undoubtedly noticed. The entire border region (defined as BorderLeft, BorderTop, etc., fields in the Window structure) is redrawn every time a window becomes active or inactive.

The colors used are different than the DetailPen and BlockPen which specified the colors under V1.3 and previous versions. We’ll describe more about this later.

Windows can now open on “Public Screens,” which we’ll discuss later. Such windows are called “Visitor Windows.” There are several ways to open a visitor window on a public screen, including all windows which open on the Workbench screen, which are now deemed to be visitors.

Windows support a “Zoom” capability, via a “window zoom gadget” which replaces one of the old depth gadgets and an Intuition function named ZipWindow() (the author has never been really happy with the term “zoom;” or “zip” for that matter).

Intuition's "zooming" consists of storing the current position and dimensions of the window, and changing them to the alternate values previously saved for the last "zoom." So it's really a swap between "current" and "other" dimensions and position. You can specify your choice for the initial "other" dimensions and position when you open your window.

Besides ZipWindow(), other new functions which operate on windows are:

- ☐ ChangeWindowBox() – specify simultaneous position and dimension change for a window, in absolute (not "delta") coordinates.
- ☐ MoveWindowInFrontOf() – brings the function of the equivalent *layers.library* routine out to the Intuition user.

Window Refreshing Procedures. We've certainly learned a lot about refreshing window damage over the last couple of years mostly that there are some subtle problems.

In particular, the BeginRefresh/EndRefresh transaction has some serious windows of vulnerability, with failures manifesting themselves as partially-refreshed window visuals. The standard simple usage of these functions doesn't cause problems, but using more than one refresh "session" (passing FALSE to EndRefresh()), refreshing through multiple user clip-regions, and needing to hold a layer lock around the refreshing session all require that you do things just right.

All Intuition programmers, experts included, should reread the autodocs for BeginRefresh() and EndRefresh() to become advised of our current understanding of their best use in tricky situations.

Window Layer Backfill. Intuition provides access to the new "backfill hook" supported by layers. That means that you can specify a routine which will be called when damage to your window is to be filled in, rather than the default "blast to color 0."

You can provide your replacement hook to OpenWindow(), so that your window will initially be filled in by your own backfill processing.

This supports pattern backfill in windows, and even "null" backfill, which yields, for example, windows which can be opened on a custom bitmap screen without corrupting the picture already contained in the bitmap.

More Window Features. As the autodocs for OpenWindow() explain, you can specify the dimensions of the interior of your window – that is, within the border – and let Intuition add on whatever it will for the borders to arrive at the total window dimensions.

This is best coupled with the new AutoAdjust option for OpenWindow(), which grants Intuition permission to slide or shrink your window dimensions in order to ensure a legal initial position and dimension.

We used up all the Window.Flags bits, and there's a new field, Window.MoreFlags, that will have to last us for a while. The only flag yet defined for that field is not documented, tested, nor completely supported at this time, but it has something to do with the "Help" key and menus.

Changes to Screens

OpenScreenTagList() is the preferred method for passing tag lists when creating a new custom screen. The tag list method is frequently more useful for screens than windows, since so many screen attributes have reasonable default values. See the autodocs for OpenScreen() and the include file *intuition/screens.h* for all the details.

New Screen Display Modes. We support many new screen modes in V36, and the major change is that the display mode specifier is now an abstract 32-bit "ID Key" defined in *graphics/displayinfo.h*, and which serves as an index into the new Graphics library display database. See the accompanying documentation of this database and new graphics display capabilities.

Programs which perform direct and on-the-fly manipulations of screen viewports are supported as best we could. Flying mode changes, poking of viewport and view offsets, and every other trick we could detect on our development and test equipment are snooped out and accommodated.

Our goal was to ensure that most tricks that worked on the original Amiga display modes would continue to work on those old modes, but new, updated processing would be required to manipulate the new modes.

In particular, the high-order bits of the CAMG ILBM chunk are now relevant. However, we try to support old programs using new modes by arranging that a reasonable substitute for a new mode be used if an old ILBM display program tries to display a picture saved in one of the new modes.

It is also possible for a machine configured for NTSC to open a PAL screen, including a PAL Workbench screen, and vice-versa. But it is important to remember that the Workbench being in PAL mode does not imply that the default display modes are all going to be PAL. In fact, there is not currently support for "tricking" a program that assumes that default operations are PAL into opening a PAL screen on an NTSC machine, and vice-versa.

Screen Coexistence and Coercion. The original Amiga display modes can all be displayed simultaneously in separate screens (with some global considerations for interlace). Some of the new modes necessarily exclude the possibility of showing other screens in their intended mode. For example, the A2024 special modes cannot be displayed simultaneously with anything else, including each other. New "31KHz scan rate" screens require that the monitor run at a different sync rate (and beam speed) than the old modes, and you cannot drastically change the monitor scan rate mid-frame.

Intuition manages these problems first by decreeing that the frontmost screen will be displayed in its intended mode. Other screens behind it will be excluded, (i.e., not visible) if necessary, as is the case with A2024 special modes and everything else. (There might be an interesting bug in current software with this.)

Sometimes, Intuition will provide an alternate, or coerced, display mode for a screen behind so that it can coexist with the frontmost screen. In doing this, severe aspect ratio distortion can occur for the screens behind. Intuition can sometimes compensate for this by switching to a higher resolution pixel rate at the sacrifice of some colors, or by introducing interlace to shorten the vertical size of pixels.

These two operations can be independently disabled via the new IControl preferences tool.

Scrolling, Display Clip and Auto-Scroll. Direct support for hardware screen scrolling has been added by taking advantage of the flexible display capabilities of the Amiga. You may now define a raster larger than can be displayed and scroll that raster through an on-screen viewable region. In this setup, the screen dimensions reflect the raster dimensions and the on-screen region is defined by a new construct named DisplayClip. The specification of DisplayClip may be made via `OpenScreen()`, but this is typically only necessary in an overscan situation.

Note that when you specify an alternate DisplayClip region – whether it is a standard one using `SA_Overscan` or a specific value such using `SA_DClip` – the values `STDSCREENWIDTH` and `STDSCREENHEIGHT` are both relative to the DisplayClip. That is, they evaluate to its width and height, respectively. This makes it simpler to do overscan. By specifying `OSCAN_STANDARD` as the data item in the `SA_Overscan` tag and setting an appropriate width and height to `STDSCREENWIDTH` and `STDSCREENHEIGHT` you can automatically create overscan displays. In fact, you are even relieved of the burden of specifying proper `LeftEdge` and `TopEdge` values in some cases. See the autodoc for `OpenScreen()`.

Two other important scrolling features have been added. Screen and DisplayClip positions may take negative values to affect scrolling up and left from the default position. Also,

automatic scrolling to follow the mouse is now supported. There are two ways to do this: by using the `MoveScreen()` function in your application or by using direct Intuition support by setting the `AUTOSCROLL` flag in your `NewScreen` structure.

GetScreenData(). This function has served two useful roles:

- ☐ Provide information about the Workbench screen for programs about to open a window on that screen.
- ☐ Provide information about the Workbench screen for programs which wish to open a similarly configured custom screen.

If the Workbench screen is in one of the new display modes, this latter purpose is not going to be well-served, especially since the screen's viewport has only 16 bits of Mode information and the new modes must specify 32 bits. To the extent that we don't jeopardize the first purpose, there have been some intense tricks applied to this function in support of the second. Also, large scrolling Workbench screens have "tricked" programs which use `GetScreenData()` into opening excessive 2400x1200 pixel windows. You'll want to read the autodocs for `GetScreenData()` carefully, then use `LockPubScreen()` as its replacement in V36 and later.

Screen Colors. You can now specify, using the screen attribute tag `SA_Colors`, a set of color palette entries to be applied at by `OpenScreen()`. This relieves you from the burden of having to call `LoadRGB4()` for your screen, and eliminates unsightly color flashing when you do.

The initial palette for new screens is determined largely by the graphics function `GetColorMap()`, but Intuition has always overridden colors 0-3 (playfield colors) and 17-19 (sprites, and sometimes playfield) with values from preferences. Intuition now maintains 32 screen color preferences values which it will use for deeper Workbench screens and (our little secret) 15-color pointer sprites.

To apply these colors to all new screens would be incompatible, so you must specify the Boolean screen attribute `SA_FullPalette` if you want your screen color map to be initialized to all the colors that Intuition ever maintains.

Public Screens. There's a section below that describes the details of the new public screens protocols. For now, be advised that the details are contained in the autodoc for `OpenScreen()`.

DrawInfo, Pen Specifications and the New Look. We'll talk more about the new look below, but the big news is that it could not be safely applied to all custom screens. In particular, ignoring the `DetailPen` and `BlockPen` designations for window border colors was

not sound. As a result, V36 does not apply the full “3D new look” effect on custom screens unless they so specify (as described below). Instead, it uses a version of its “monochrome scheme” applied in the screen’s DetailPen and BlockPen.

Rendering for the new look uses information encapsulated in the DrawInfo structure of each screen. You may acquire a pointer to this data by calling GetScreenDrawInfo(). In that data, there is an array of pen numbers used for different drawing purposes, including 3D shine and shadow, highlighted window borders, text, text in highlighted borders and so on.

You may specify replacement values for those pen specifications (sometimes called “pen spec” in the documentation) using the screen attribute SA_Pens. You may specify many, a few, or zero pens using this tag. Intuition will take the presence of this tag to mean that you are a new custom screen, aware of the new look, and will set the DRIF_NEWLOOK flag in your screen’s DrawInfo structure, and give windows in your screen the complete full-blown 3D new look treatment.

Miscellaneous Screen Changes. Here are a few small items:

- ☐ You can pass the pointer to a longword using the SA_ErrorCode tag, and Intuition will provide in that longword a detailed error code to supplement the NULL screen pointer it returns in the event of failure. The defined error codes today are:

OSERR_NOMONITOR	- named monitor spec not available
OSERR_NOCHIPS	- you need newer custom chips
OSERR_NOMEM	- couldn't get normal memory
OSERR_NOCHIPMEM	- couldn't get chipmem
OSERR_PUBNOTUNIQUE	- public screen name already used
OSERR_UNKNOWNMODE	- don't recognize mode asked for

You should also assume that there will be more error codes defined in the future, and allow for codes your program doesn’t understand.

- ☐ SCREENQUIET is really quiet. The menu bar layer no longer blasts out a clear region in the custom bitmap of a SCREENQUIET screen. It is created with a “null” layer backfill hook. SCREENQUIET screens are finally completely “quiet.”
- ☐ Multiple System Font Selections. When opening a screen, you can specify a font as a pointer to a TextAttr structure. If you provide none, your screen’s default font will be the current graphics default font (GfxBase.DefaultFont), which is maintained by old and new preferences to be some safe, non-proportional conservative font that all programs should be able to handle even if it is no longer only topaz.font size 8 or 9.

We also maintain via preferences a new font selection by the user, the “default (fancy) screen font,” which is often proportional-space and taller than normal. You can select this font to be your screen font (as does the Workbench screen) by using the SA_SysFont attribute tag and providing the ti_Data value of 1.

When a proportional font is used as a screen font, though, we run into problems when the windows opening on that screen, especially the Workbench screen, inherit the screen font in their rastport. We’ve found only a few text-oriented programs that work when they are surprised with a proportional font. So, we modify an old rule to have an exception:

The windows opening on a screen will have as their default font the font of the screen, unless when the screen opened its font was specified as this new preferred screen font. In that case, newly opened windows will have their RastPort initialized with the system (monospace) default font.

Gadgets

There is change in the behavior of NewModifyProp() to minimize redraw flashing when a gadget is updated. There is also a separate document about custom gadgets and object-oriented (boopsi) gadget classes, group gadgets, and so on. There’s a list of the pre-defined gadget classes in the “boopsi” section below. The rest of this section is about string gadgets.

String Gadgets. String gadgets have numerous enhancements, most specified by additional data defined in the StringExtend structure, defined in the include file *intuition/sghooks.h*. The most significant work was done in the support of proportional fonts.

The new specifications include the font, the pens, optionally different pens for use when the gadget is active, new editing modes including REPLACE mode and FIXEDFIELD mode.

You may also provide a hook which can be used to intercept Intuition’s default input processing for the gadget, allowing custom editing. See the example StrDemo on the DevCon disks, and the contents of *intuition/sghooks.h* for details.

Note that late in the development of V36, two new features were added to this custom editing hook:

- ☐ Your hook is now called to process mouse click input, not just typing.
- ☐ Your hook is passed an “edit operation code” so you don’t have to guess whether Intuition’s intentions were to delete or insert a character, delete to beginning, or whatever.

There is a function which can be used to replace the global editing processing that Intuition does by default for all string gadgets. This function is `SetEditHook()`, but this feature has never been exercised to the knowledge of the author. This function should therefore not be used in a commercial product until further notice. If anybody ever gets it working to provide “vi-style” editing for string gadgets, though, the author would greatly appreciate a copy.

For string gadgets which inherit a font larger than they were prepared to contain, Intuition will “fall back” to Topaz 8.

String Gadgets also have the ability to “filter” control characters (which are even invisible in some fonts) from being entered into the text. This is globally established as a preferences item in the IControl preferences tool. When the filter mode is on, certain control characters are used as editing commands, consistent where possible with the same keystrokes used to edit input for the console. When filter mode is on, the user can still enter the occasional control character into the text of a string gadget by holding down BOTH the control and left-Amiga keys when typing a letter.

Requesters and System Requesters

There are some minor enhancements to application requesters (brought up by the `Request()` function or as a `DMRequest`), but the major work involves system requesters which are gotten using the old functions `AutoRequest()` and `BuildSysRequest()` and the new functions `EasyRequest()` and `BuildEasyRequest()`.

The new `EasyRequest` functions provide a “printf-style” automatic formatting of system requester text and gadget labels, support more than two requester gadgets, and perform automatic font-sensitive layout of text and gadgets. Both types of system requesters are now implemented in a fancy “new look” design, which, in the case of old-style `AutoRequesters`, happens by Intuition ignoring a large amount of the layout specifications you provide. See the autodocs and the Intuition DevCon example *EasyReq* for several ways you can use `EasyRequesters`.

The functions `AutoRequest()` and `EasyRequest()` are synchronous, that is to say, these functions don’t return until the requester has been “satisfied” either by user input or other IDCMP traffic (e.g., `DISKINSERTED`).

For each, there are corresponding functions `BuildSysRequest()` and `BuildEasyRequest()` that return to you a pointer to a window which contains the system requester. You may process the input to that window yourself to gain more control over the request. An example might be timing out the request or, like the `IPrefs` daemon, watching for circumstances which warrant an automatic “Retry.”

The input processing for these windows is not completely trivial, but it is made very easy by a new function `SysReqHandler()`. See the autodocs for some examples of fancy processing made easy.

System requesters have, since V33, popped the screen they appear on to the front if it was not already frontmost. This is to alert the user. With V36, we will subsequently pop that screen to the back (NOTE: not “back to where it was”) when the requester is satisfied, provided that the user did not perform manual screen depth arrangement or open or close other screens in the interim.

General Requester Changes. Here are some changes made to general requesters. Some of these were introduced to support the new look of system requesters.

- ☐ POINTREL now works with `Request()`, providing a simple method for requesters to appear centered in an application window.
- ☐ The new flag `SIMPLEREQ` specifies that you want your requester to be in a simple refresh layer. Recommended for saving chip memory and avoiding a large transient requirement for chip memory when the requester opens.
- ☐ A new flag `USEREQIMAGE` indicates that your requester has a valid pointer to a linked list of Intuition Images in the new `ReqImage` field. This list of images will be automatically drawn by Intuition when the requester comes up or needs to be redrawn. Using boopsi image classes such as `fillrectclass`, `frameiclass`, and `itexticlass` is how the system requesters fancy new look is maintained without special refresh code. You can do almost anything with a boopsi image, and hence have some very fancy requesters which are automatically maintained.
- ☐ The new flag `NOREQBACKFILL` specifies that your requester layer should have a “null” backfill hook, so that it is not cleared before your fancy images described above are drawn. This is why system requesters don’t flash before the stipple pattern is drawn.
- ☐ There is a limit to the number of requesters that Intuition will support for a window, if that window is capable of being moved or changed in size. For V36, that limit is 8.

IDCMP and Event Communication

New IDCMP Classes.

- ☐ `CHANGEWINDOW` is sent (via IDCMP or through the console) when your window position or dimension changes, including changes by new functions `ChangeWindowBox()`

and ZipWindow(). It generalizes both SIZEWINDOW and the proposed MOVEWINDOW messages.

- ☐ IDCMPUPDATE is sent by boopsi gadgets as a more direct connection to the internal state changes of custom boopsi gadgets. See the Intuition boopsi examples for comparisons to using this and using GADGETUP and FOLLOWMOUSE messages. In this message, the qualifier IEQUALIFIER_REPEAT indicates “interim” reports, such as a button repeating while held down or a slider being dragged vs. their being released. All other qualifiers are meaningless.
- ☐ MENUHELP is sent in place of MENUPICK if the user presses the Help key while operating a menu. This is not yet officially supported and requires that an undocumented flag be set.

Other Changes. There are now automatic limits to the number of MOUSEMOVE or repeat-key IDCMP messages that Intuition will send to your window before seeing some replied. These limits can be independently set when you open your window. The mouse queue limit can be changed using the function SetMouseQueue(). There is as yet no interface for changing the repeat queue limit on the fly. You can set these limit values to -1 to get effectively unlimited queue or backlog of these messages.

The middle mouse button is now supported and is transmitted as new values of the *IntuiMessage.Code* field for class MOUSEBUTTONS.

If you specify the VANILLAKEY IDCMP Flag, you will receive as always VANILLAKEY messages for all keyboard downstrokes that translate into single byte character code. But now, if you also specify the RAWKEY flag, you will receive RAWKEY reports for those downstrokes which do not map to a single byte. This makes it easy to process function keys, the cursor keys, or the Help key, without sacrificing the convenience of VANILLAKEY messages.

Verify Functions Abort. The most common system deadlocks were caused by deadly embrace around the MENUVERIFY, SIZEVERIFY, and REQVERIFY messages. Intuition used to wait dead for your application's reply to these messages, and if your program was for some reason waiting for Intuition input before it would get around to making such a reply, the system would deadlock.

Now, Intuition will time-out these verification transactions after user-specified interval, and abort them when the user releases the relevant mouse button, where that makes sense. Your program may respond affirmatively to a message like MENUVERIFY at some time after Intuition has decided to abort. Your program will receive whatever termination message it

expects subsequently, even though that might mean sending both a MENU PICK message (which your program expects after an affirmative response) and a MOUSEBUTTONS/MENUUP message (expected when you cancel a MENUVERIFY request).

There might be some quirks that can result – in fact, there are known problems (and tested fixes waiting) with the speed of these transactions and “hanging menu strips” – but we’ll have no more deadlocks caused by this frequent culprit.

Images

We’ve introduced a few new functions for images, but most of them aren’t too useful for normal old Intuition images. All the excitement is about new custom images created from boopsi classes. There are examples in the boopsi subdirectory of the Intuition DevCon demo directory.

- ☐ DrawImageState() is an enhanced DrawImage() that delegates to the image the type of rendering that should be done to represent a selected, disabled or inactive state. This is useful for fancy images attached to fancy gadgets.
- ☐ EraseImage() clears out (using the new EraseRect() graphics call) the region occupied by an image.
- ☐ PointInImage() asks an image whether a particular X/Y point coordinate is contained in the image region. This encapsulates the intelligence of things like the BOOLMASK feature of V33 gadgets.

There are some very useful boopsi image classes which you can use to create new images for your use in gadgets, menus, requesters, or just to pass to DrawImage() or DrawImageState(). See the boopsi section for a list.

Public Screens

Many developers and power users have requested support for “shared custom screens.” They wish to bring up utilities such as a text editor on an application custom screen such as a terminal emulator. A similar request has been made for “multiple Workbench screens” so that Workbench application windows can be grouped in separate workspaces and display modes best suited for the different Workbench applications that may co-exist.

It turns out that both of these requests can be satisfied by new support in Intuition for what we call “public screens.” The complete solution will have three parts: support within Intuition, a screen manager utility and, of course, applications which open the new public

screens and their “visitor” windows.

When opening a screen under the new system, you may declare your screen a public screen by specifying a public name string for it. You may also register your task ID and a signal if you want to be notified when the last visitor window on your screen closes. A new data structure, PubScreenNode, will be installed on a system list with links to your public screen.

A public screen has a single mode bit named PSNF_PRIVATE which indicates whether the screen is available for public use. A public screen is initialized as private so you can open your personal application windows (like a backdrop window) before you make the screen officially public.

Once you have a public screen, visitor windows can be opened on it in several ways. If no screen name is provided, the visitor window will open on the default public screen. If a name is provided, the visitor window will open on the named screen if available. Otherwise a default is used. A visitor window can also be opened by using a pointer to the public screen obtained with LockPubScreen().

There are two global mode bits which also come into play when a visitor window is opened on a public screen. If the global mode SHANGHAI is set, Workbench application windows will be opened on the default public screen. A second global mode bit is POPPUBSCREEN which indicates that a public screen is to be moved to the front when a visitor window opens on it. Both the SHANGHAI and POPPUBSCREEN mode bits are intended to be readily controllable by the human user – likewise the DefaultPubScreen.

One example of an application using visitor windows will be the public screen manager utility. Its role will be to display the current public screens, allow selection of the default and setting of the global modes SHANGHAI and POPPUBSCREEN. An important elaboration will be this program’s ability to open named public screens listed in a configuration file at the user’s command.

Other special applications of public screens are possible. One operation now supported by an Intuition function is “jumping between public screens.” In this way an application can move between screens in response to a gadget or menu item, or in response to a hotkey.

An example program, PubSC, is included on the DevCon disks. This program demonstrates the primitive operations now supported by Intuition. It has been greatly improved since early alphas, and is actually a first cut at the screen manager utility. Further information for this topic can be found in *intuition/screens.h* and in the autodoc sections for OpenWindow(), OpenScreen() and the new support functions.

Here's a brief list of the functions:

<code>PubScreenStatus()</code>	-- Converts a public screen to private or non-private status. Fails if you can't make a screen private because it currently has visitors. You do this before you close a public screen.
<code>SetDefaultPubScreen()</code>	-- Establishes a given public screen as the default.
<code>GetDefaultPubScreen()</code>	-- Returns the name of the default screen for use by the screen manager utility (the name is not needed by normal applications).
<code>SetPubScreenModes()</code>	-- Sets new values for the public screen global modes.
<code>NextPubScreen()</code>	-- Helps an application cycle its window between the currently available public screens. Note that the Workbench screen is a public screen, and will be included in the rotation.
<code>Lock/UnlockPubScreen()</code>	-- Allows to you to determine that a public screen exists, and to insure its existence while you open a visitor window on it. This function also serves as an improvement over <code>GetScreenData()</code> , as a fortunate side-effect.
<code>Lock/UnlockPubScreenList()</code>	-- Affords protection while you COPY the Intuition public screen list quickly. This is intended for use by the public screen manager utility only.

Intuition Classes, "boopsi"

The term "boopsi" is a lower-case acronym for the "Basic Object-Oriented Programming System for Intuition." The key word here is "Basic."

Boopsi supports custom gadgets, custom images, grouping of gadgets, and interconnections between them. It is also very open-ended, and can be extended in many directions for general-purpose use.

You can invent boopsi classes which implement the behavior for the objects they define and create. Classes can be private, which means the code for them is linked into your program and you don't have to worry about names and attribute IDs colliding with other defined classes. Classes can also be public, accessed only by their unique names. In that case, their attribute IDs must be picked carefully not to conflict with any others. This will require a registration procedure to be implemented by Commodore Applications and Technical Support.

There is a separate document on boopsi, to which we refer the reader for all details. We list here only a summary of the pre-defined public classes implemented in Intuition, grouped by their "baseclass," that is, their inherited ancestor closest to the root.

rootclass

Ancestor to all boopsi classes, the only class without a superclass. Implements basic object memory allocation and nodes.

gadgetclass

Ancestor to all boopsi gadget classes, understands most gadget attributes.

propgclass

A very friendly way to use one-dimensional proportional gadgets, including a better way to look at Body and Pot values, and interconnection capability.

buttonclass

A (repeating) boolean command button.

frbuttonclass

A high-powered button for the "new look" which knows how to frame its general contents (label) with an embossed box.

strgclass

An easy interface to the new extended string gadgets, with many attributes easily defined. Supports interconnection.

ggclass

Group-gadget class, this implements a "composite" gadget which has other gadgets as its members. The DevCon demos illustrate how you can collect several related gadgets in one of these group gadgets and have them all interconnected, sending your application an IDCMPUPDATE message only when something of interest has changed.

imageclass

The common ancestor of boopsi image classes. Implements most image attributes but draws nothing.

frameiclass

An embossed or recessed rectangular frame, rendered in the proper DrawInfo pens, and with enough intelligence to bound or center its contents.

itexticlass

A specialized image class implementing something similar to IntuiText. This is used by Intuition for forcing the new look on AutoRequesters, and for supporting System Requester button gadgets which don't need a full IntuiText allocated for them.

sysiclass

A class of premade "system images" including all the images for the system gadgets, and the gadtools checkmark and button glyphs. These images can be created to specified size, scale, and will support all the new look voodoo.

icclass

A very simple interconnection node. Maps one set of attributes (understood by

something notifying other objects of changes) to another set of attribute tags (as understood by the object being notified). The heart of boopsi interconnection, often found hanging in a broadcast list.

modelclass

A specialization (subclass) of icclass, which adds the notion of a broadcast list. You notify these things of a change, and they broadcast that to their whole list, with attribute IDs being mapped independently by the ic's hanging on the list to what the other interested objects need to hear.

The New Look

The new 3D “embossed” look of Intuition window borders and gadgets is the most obvious major change to Intuition. It represents a compromise between our compatibility goals and “giving a little to get a lot” in the way of major aesthetic rework.

There are actually a couple of different “looks” Intuition will apply to a given screen and all the windows within that screen.

- ☐ The full-blown embossed new look, as found on the Workbench screen. This requires that we make certain assumptions about which pens to use for different things. We don't default to this look (anymore) for custom screens. Only applications that run on the Workbench will be confronted with these changes at first.
- ☐ The monochrome new look. In a single bitplane (two color) screen, there isn't a whole lot of 3D embossing you can get. We fall back to a simpler two-color scheme, with specially designed system gadget images suitable for a monochrome presentation.
- ☐ “Compatible” custom screen new look. Normal custom screens are rendered in a two-color version of the monochrome new look. The pens used for this rendering are the initial screen DetailPen and BlockPen. All windows are rendered in these same colors.

If you want your custom screen to be given the full-blown new look, you pass the SA_Pens screen attribute tag to OpenScreen().

DrawInfo Structure. All the information needed to render the new look is packaged in a data structure called DrawInfo associated with each screen. You manage a pointer to it by calling GetScreenDrawInfo() and FreeScreenDrawInfo().

We'll discuss the fields in DrawInfo now. See also the structure and flag definitions in *intuition/screens.h*.

dri_Version - this field is initialized by Intuition to the value of the constant DRI_VERSION defined in the version of screens.h that Intuition was compiled with. That means that if you see that dri_Version is greater than or equal to the value of DRI_VERSION that your program was compiled with, you can be assured that any new fields in DrawInfo that are defined are being supported by Intuition.

dri_Font - this is a pointer to the screen's (open and ready to use) font. Do not assume that this font will remain open after you call FreeScreenDrawInfo().

dri_Depth - the depth of the screen's bitmap, as provided to OpenScreen.

dri_Resolution - a pair of numbers gotten from the display info database for the (initial) display mode of the screen. These support simple resolution-independent constructs, such as drawing square boxes. The values for NTSC HIRES+LACE pixels are X = 22 and Y = 26.

dri_Flags - there is only one flag now defined, DRIF_NEWLOOK. If this flag is set, the attribute SA_Pens was provided to OpenScreen(), and the screen is to get the full-blown 3D new look treatment.

dri_NumPens - this is the number of pens defined in dri_Pens.

dri_Pens - this is an array of drawing pens used for different purposes. The number of entries is specified above, and the array is terminated by 0xffff (~0).

The pens defined are:

detailPen	- initial value of Screen.DetailPen
blockPen	- initial value of Screen.BlockPen
textPen	- text on normal backgroundPen
shinePen	- bright edge on bas-relief
shadowPen	- dark edge
hifillPen	- active window fill
hifilltextPen	- text over hifillPen
backgroundPen	- always 0 for now
hlighttextPen	- highlighted text, against backgroundPen

Elements of the New Look. Here is a discussion of the things which make up the new look for V36.

Default pens are defined for multi-color screens and monochrome (single-bitplane) screens. The defaults are:

Multi-color screens (using four colors for new look)

detailPen	= 0
blockPen	= 1
textPen	= 1
shinePen	= 2
shadowPen	= 1
hifillPen	= 3
hifilltextPen	= 1
backgroundPen	= 0
hlighttextPen	= 3

```

Monochrome screens (using four colors for new look)
  detailPen      = 0
  blockPen       = 1
  textPen        = 1
  shinePen       = 1
  shadowPen      = 1
  hifillPen      = 1
  hifilltextPen  = 0
  backgroundPen  = 0
  highlighttextPen = 1

```

Old-style custom screens will get defaults the same as monochrome screens, with “1” and “0” replaced with the initial value of the screen’s BlockPen and DetailPen, respectively.

Window Borders are filled in completely, to the (compatible) dimensions defined by Window.BorderLeft, Window.BorderTop, Window.BorderBottom, and Window.BorderRight. The outer and inner edges of the borders are drawn in a combination of shinePen and shadowPen, and the rest of the border is filled with backgroundPen or hifillPen, depending on whether the window is inactive or active.

Window title text is rendered in textPen and hifilltextPen, again depending on whether the window is inactive or active, respectively.

After refreshing the borders, Intuition refreshes all border gadgets, then draws the outline edges of the border again, to win all “border wars.”

A gadget is determined to be a “border gadget” first by the presence of one of the gadget Activation flags **RIGHTBORDER**, **LEFTBORDER**, **TOPBORDER**, and **BOTTOMBORDER**. Applications were not rigorous in setting these flags, so now Intuition “sniffs out” gadgets which intersect with the border, and sets a private flag bit so it will know to refresh these gadgets, too.

Gadgets in the border consist of both system gadgets and application gadgets. The system gadgets were replaced by new boopsi buttonclass gadgets, using images from sysiclass. These gadgets and images know how to draw themselves in a 3D look or in monochrome, and are sensitive to whether the window is active or not.

Application gadgets in the border are not generally that advanced, and they are not drawn differently, except for proportional gadgets.

Proportional gadgets, especially those with **AUTOKNOBS**, are given a completely new treatment for the new look, but unchanged (we hope) for other uses. A proportional gadget gets the new look if either:

- ☐ the proportional gadget intersects the window the border.
- ☐ the new flag PROPNEWLOOK is set in PropInfo.Flags.

The images used for system gadgets respect several of the new image draw states defined in *intuition/imageclass.h*, specifically:

IDS_NORMAL	- what old DrawImage() will yield
IDS_SELECTED	- for selected gadgets
IDS_DISABLED	- for disabled gadgets
IDS_BUSY	- for future functionality
IDS_INDETERMINATE	- for future functionality
IDS_INACTIVENORMAL	- normal, in inactive window border
IDS_INACTIVASELECTED	- selected, in inactive border
IDS_INACTIVEDISABLED	- disabled, in inactive border

The screen title bar is only partially converted to the new look for V36; it has a new depth gadget. The final design will be implemented later.

System Requesters. Requesters created by AutoRequest() and EasyRequest() also get a new treatment. They are built out of boopsi gadget and image classes, including these:

- ☐ gadgets are from frbuttonclass, which uses the image class frameiclass shared between all gadgets but stretched suitably for each when rendered.
- ☐ requester body text is managed by an itexticlass boopsi image.
- ☐ the background of the requesters is a fillrectclass object.
- ☐ the “sunken” bevelled region for the text is another frameiclass image.

Menus

There was very little enhancement to menus for V36. No pop-up, tear-off, or pie-shaped menus. Sorry. There were a few things, though:

- ☐ ResetMenuStrip() is a new function that serves as a quicker version of SetMenuStrip() that can be used only under certain circumstances. See the autodoc.
- ☐ Menu command-key equivalents are processed through the default keymap and use a correct algorithm for caseless comparison that doesn’t break down on “international” characters.

- ❑ The NextSelect cycles that could prune multiple selections from your MENUPICK list have been eliminated. There is still no solution provided to the fact that the NextSelect chain can be severed if the user operates the menus before you're done walking the chain.

Miscellaneous Items

Here's a grab-bag of items that can be briefly mentioned:

- ❑ Alerts are now green if recoverable, use their own View structure, and are always in the topaz 8 font. They don't clobber each other if DisplayAlert() is called more than once, and they never smear the sprite any more.
- ❑ The intended function for taking over the system, StealI(), was not implemented. I'm really sorry about this one. Maybe next time.
- ❑ Intuition supports taller and deeper system sprites through the new Preferences methods. Our "official" support doesn't extend to attached or very tall sprites, but you can use them if you know how . . .
- ❑ RastPort protection. Intuition is now much better about leaving the various RastPorts it uses undisturbed, except for side-effects of its use that should be felt. As an example, PrintText() will not leave any changes to font, pens, or draw mode in the rastport it is passed, but it will update the current text position fields, cp_x and cp_y.
- ❑ The rectangles drawn for interactive dragging or sizing of windows are now not so terribly clobbered by collisions with the video beam.
- ❑ Intuition's uses of DisplayBeep() go through the library vector, so if you SetFunction() it to be an audible noise, you'll pick up Intuition's uses in string gadget editing, too.
- ❑ Mouse emulation is done by adding input events through the input.device (and the command IND_WRITEEVENT). The old method was to provide an event of class IECLASS_POINTERPOS, and to specify within that event absolute or relative coordinates in some view-relative "hires-lace coordinates." That wasn't sufficient for new super hires modes, and isn't convenient for tablet-driver authors, so we added a new input class (you guessed it) IECLASS_NEWPOINTERPOS. This supports several methods of specifying mouse movement commands, including "tablet-oriented" range/value coordinates within the overall mouse movement boundaries, and "screen pixel" coordinates, which makes it trivial to position the mouse over some point in your window or screen.

New User Controls

There are some new user interface features that will affect the end user. Some of them, such as string gadget editing control chars, we discussed above. Here are some more:

- ☐ The Screen Menu Snap feature brings the upper left corner of a scrolling screen into view when operating menus. If you hold the left-Amiga key down when you press the menu button, the screen will stay at the snapped position when you release the button.
- ☐ Command keys for depth arranging screens (left-Amiga M and N), for satisfying system requesters (left-Amiga B and V), and the qualifier for dragging screens when the mouse is not over the screen drag bar (left-Amiga default) can be set to different values in the IControl preferences tool.
- ☐ The screen depth keys used to mean "Workbench to front" and "Workbench to back." Now they mean "Workbench to front" and "Frontmost to back" which lets you cycle through the screens from the keyboard.
- ☐ Keyboard methods to move the mouse (amiga-Arrows) are now much better, including moving exactly one pixel per keypress (non-repeating, and in the pixels of the active window's screen), and have a slow, predictable ramp-up as you hold down the key and it repeats. While the cursor keys are repeating, you can tap the shift key for a burst of acceleration. Try it.
- ☐ Interactive window dragging and sizing can be aborted by pressing the menu key during the operation.
- ☐ There is an optional mouse accelerator built into Intuition. It is considerably leaner than the one implemented in earlier alpha and beta releases.
- ☐ The window and screen depth gadget pairs have been consolidated into single gadgets. When the screen or window is frontmost, selecting the gadget sends it to back. Otherwise, it is brought to the front.
- ☐ The window zoom gadget replaces one of the old depth gadgets.

About the Demos and Examples

Many of these demos are familiar from previous devcons, but there are some pretty good new ones, too. See the README file for the standard logical volume "Assigns" that are used to compile and link these examples. Some have a makefile, others have compilation

instructions in the source file or the README.

PubSC – Public Screen Demo Program. This program has been completely rewritten since the early alphas. It is now an Intuition-based program with command and state gadgets. It includes some obsolete but nice gadget- and image-building techniques and also adapts itself to fonts of various sizes. It is itself a “visitor” window, and will open on the default public screen.

The command gadgets included are Close Screen, Make Default, Open New and Jump. Each command may also be invoked by typing the initial character of the gadget label.

The Close Screen gadget attempts to close the current screen – that is the public screen that the PubSC window opens on. It will not attempt to close the Workbench screen. The gadget reports failure if it cannot close the current public screen because other windows are open on it.

The Make Default gadget requests that the current screen be made the default public screen. This may be selected for the Workbench screen.

The Open New gadget will open a new public screen, named “Phred.” It will report if screen “Phred” is already open. If you want to change the mode or name used you will have to modify the source.

The Jump gadget will cause the PubSC window to “jump” from one public screen to the next. This is useful if you have a lot of public screens.

The state gadgets establish global Intuition public screen modes. They are named “Auto Pop-to-front” and “Hijack Workbench.” The former determines whether a public screen will be moved to the front when a window opens on it. The latter determines if application windows which are intended for the Workbench screen are diverted to the default public screen. This allows you to work in multiple Workbench environments, partitioning your projects and minimizing clutter. In addition to the state gadgets, PubSC has text displays showing the names of the default and current public screens.

The source to PubSC is extremely important for understanding public screens. Especially important are the procedures for locking a public screen such as the Workbench, adapting a window layout to the screen’s parameters and font, and opening a visitor window.

The PubSC example is font-independent but not yet resolution-independent although the source shows what needs to be done. PubSC runs from CLI.

ScDemo (formerly SC, also ScreenDemo) – Screen Test Program and Demo. This program has been substantially rewritten to take advantage of the simpler new methods for opening screens in new modes. The new version also establishes overscanned screen dimensions. The program is run from the CLI. Typing “?” will give you the available options. Open a few screens. Typing “w” opens a wide screen which you can scroll by dragging the scroll bar or (by pressing left-Amiga-select). Try a productivity screen. With several screens open, try the new cycling command left-Amiga-M. You can close all the screens by typing “q” for quit or by typing “f” for free screens. Read the source file *sc.c* to see the methods for opening the various screen modes and dimensions.

StrDemo – Demonstration Of New String Gadgets. This program is run from the CLI. It opens a new screen, a window, and several enhanced string gadgets. This program is most impressive if you have ruby 12 and 8 in your fonts: directory. Besides being a test for proportional fonts and a demo for some of the new capabilities, this program also demonstrates a custom editing hook. Note that “TAB” and “Shift-TAB” will activate gadgets in succession. The second gadget from the top also has an editing hook. Pressing up and down arrow keys cycle through a list of pre-set choices. Selecting the close gadget in the window terminates the program.

CustGad – Custom Gadget Demo. This program provides a custom gadget example named “Dial Gadget.” It also uses some floating point calculations. If we had used the IEEE math libraries, there would be a problem, since the only task that can use the IEEE libraries is the same task that opened them. This illustrates an important point. Remember that with custom gadgets, the code you provide will be run as different tasks, including the input.device. It is important to limit the activity that takes place in the gadget itself, and make more complex operations occur in response to a GADGETUP or GADGETDOWN message.

EasyReq – Example of Using EasyRequest(). Type “EasyReq” from the CLI and the resulting requester will tell you how you can experiment with fancy formatting strings and arguments. When using this demo, all arguments passed to EasyRequest() are strings (*argv[]), so the only “printf” formatting commands that makes good sense for the demo is “%s.”

The boopsi Demos Subdirectory. There is a suite of boopsi examples. One sequence of demos (demo1, demo2, demo3, . . .) solves the same problem with increasingly sophisticated use of boopsi gadgets and classes, but with decreasing work required by the application “main” programming. There are examples of public and private boopsi class implementations for gadgets and images, and examples using frame images which bound and center their contents. See the README file in that directory. You will want to have a 9600-baud terminal attached to your Amiga for kprintf output since gadget and image class code is generally not run as your application’s task. You’ll certainly need kprintf output to develop such classes anyway. ♦

1

2

3



The Gadget Toolkit

by Peter Cherna

The Gadget Toolkit is an Amiga shared library designed to simplify the task of creating elegant and efficient user interfaces under Intuition. It offers a flexible and varied selection of gadgets and menus to help programmers through what used to be a difficult part of their task.

I. Introduction

Intuition, the Amiga's graphical user-interface, is a powerful and flexible environment. To its credit, Intuition allows a software designer a great degree of flexibility in creating dynamic and powerful user-interfaces. The drawback of this flexibility is that programming even straightforward user-interfaces fairly involved, and certainly difficult for first-time Intuition programmers. If we consider the saying "Simple things should be simple and complex things should be possible", it is clear that Intuition delivers better on the latter score than on the former.

What we have attempted to do with the Gadget Toolkit (GadTools) is to harness the power of Intuition and deliver easy-to-use high-level chunks of user-interface. While GadTools doesn't pretend to answer all the possible user-interface needs of every application, it works fine alongside those sections of the operating system that are designed to meet special needs, such as Intuition's already-familiar gadgets and its new BOOPSI object-oriented custom gadget system. By meeting most of the user-interface needs of most of the applications, GadTools should greatly simplify the problem of designing user-friendly software on the Amiga.

A key benefit of GadTools is its standardized and elegant look. All applications that use GadTools will share a similar appearance and behavior. Users will appreciate a sense of instant familiarity even the first time they use your product. As well, the expected prevalence of GadTools means that a developer whose needs extend beyond what GadTools offers will create custom gadgets which nonetheless share the look and feel of GadTools.

GadTools provides a significant degree of visual consistency. Certainly this is true across multiple applications that use it, beginning with several that are included with AmigaDOS 2.0, such as the Preferences editors, Workbench's "Information" window, and the Commodities Exchange. There is internal consistency between different elements of GadTools, too. You will notice that the look is very clean and orderly. Depth is used not just for visual embellishment, but as an important cue. While the user is free to select something inside a "raised" area, the "recessed" areas are informational only, and clicking in them has no effect.

It is very important to realize that GadTools is not amenable to "creative" post-processing and tweaking by programmers who are looking to achieve something a little bit different or beyond what GadTools offers. Developers are warned to play by the GadTools rules. Only in this way may GadTools grow and improve without hindrance, even allowing future features to automatically appear in your software when reasonable.

Most developers are more concerned with and skilled in their areas of interest (which determine what type of product they are developing) than they are in other aspects needed to complete their product. The author of a three-dimensional rendering package would reasonably be more interested in better algorithms than in tweaking menus and gadgets. Understandably, this has led to programs with unorthodox or awkward user-interfaces. GadTools will make the path of least-resistance in user-interface design a path that will present the user with a consistent interface of high quality. A motto for GadTools might be "User-friendly, and programmer-friendly too!"

II. Ingredients of GadTools

GadTools consists of a body of routines to create, manage, and delete various useful kinds of gadgets, such as buttons, sliders, mutually exclusive buttons, and scrolling lists. As well, GadTools allows you to easily create, lay-out, and delete Intuition menus.

To illustrate the kind of flexibility, encapsulated power, and simplicity that GadTools offers, we can look at the GadTools slider gadget, which would be used to indicate and control the level of something, for example volume, speed, or color intensity. While Intuition proportional gadgets deal in the arcane Body and Pot variables to control the slider knob's size and position, with GadTools the programmer directly specifies the minimum and maximum levels of the slider, as well as the current level. A color slider would have a minimum level of 0, a maximum of 15, and the current level might be 11. To simplify your event-processing, GadTools only sends a message when the knob has moved far enough to cause the slider level (expressed in your terms) to change. If a user were to slowly drag the knob of this slider all the way to the right, the program would only hear messages for levels 12, 13, 14, and 15, with an optional additional message when the user released the mouse-button. To change the current level from within your program is as simple as calling a function and saying that the new level is to be (say) 5. As a final point, the slider is very well-behaved. When the user releases the mouse-button, the slider immediately snaps to the true position of that level. If a user wants to set his background color to light gray (say red = green = blue = 10), all three color sliders will have their knobs in precisely the same position, instead of anywhere in the range that means "ten".

III. Introduction to Tags and Tag-based Functions

The Amiga has hundreds of different system structures, each laden with fields and flags. There must be thousands of intriguing bits of information that can be read or written. However, setting a flag or field in a structure is far from ideal from the system's perspective. This is easily felt, such as when an Intuition programmer discovers that it can be a lot harder to disable a gadget than to simply set its GADGDISABLED flag. Often, the more complex the object you deal with, the less appropriate it is to write into structures.

Sometimes, a better model for many areas of the system is one in which all transactions are function-based. To change an attribute, the programmer must call a function that sets that attribute to its desired value. Rather than provide a different function for each of the multitude of attributes, a small number of "tag-based" functions work quite nicely. Each attribute is specified as a TagItem, which is a unit that has a "tag" which identifies the attribute being set, and a corresponding "data" element which contains that attribute's value (which might be a number, a pointer, etc.).

If we allow such functions to accept a variable number of TagItems, we then have a very powerful set of flexible and extensible functions. The caller only specifies those attributes that he needs, and omits those he does not. Many attributes have reasonable defaults if they are left out, though some attributes may be required and the function will fail if they are omitted.

A function that accepts tags takes a pointer to an array of tag-and-data pairs. For example, the sequence:

```
struct TagItem mytaglist[] =
{
    FRUIT_Kind, "Peach",
    FRUIT_Number, 12,
    FRUIT_Juicy, TRUE,
    TAG_DONE, 0,
};
CreateFruitA( mytaglist );
```

This might give you a dozen juicy peaches. If you were to omit the FRUIT_Number TagItem, you would get the default, which might be a single peach. If you omitted the FRUIT_Kind, the function might reasonably fail, since there might be no concept of a default kind of fruit.

The TAG_DONE tag is a special system value to indicate the end of a tag list and is required even if it would be the only TagItem.

From any programming language, tag-based functions accept an array of TagItems that is terminated by the TAG_DONE tag. From some languages such as C, there are alternate entry points (stubs in amiga.lib) which accept a more convenient representation in which the tags are built on the caller's stack. These stack-based functions typically bear the "natural" name, while the array-based ones often have a post-pended "A" in their function name. Thus, we have the stack-based equivalent of the above, namely:

```
myfruit = CreateFruit(  
    FRUIT_Kind, "Peach",  
    FRUIT_Number, 12,  
    FRUIT_Juicy, TRUE,  
    TAG_DONE );
```

(Since processing will stop at the TAG_DONE tag, no corresponding data value need be supplied.)

All GadTools tags begin with a leading "GT". In general, they also have a two-letter mnemonic for the kind of gadget in question. For example, slider gadgets recognize tags such as "GTSL_Level". The GadTools tags are defined in <libraries/gadtools.hli>. Certain GadTools gadgets also recognize Intuition tags such as GA_DISABLED and PGA_FREEDOM, which can be found in <intuition/gadgetclass.h/i>

For more information on tags and tag-based functions, you may wish to browse the autodocs, include files, examples, and other documentation for utility.library, as well as for gadtools.library and intuition.library.

IV. GadTools Gadgets

The heart of GadTools is in its ability to help you create and manipulate a sophisticated and varied array of gadgets. GadTools supports the following kinds of gadgets:

- ☐ Button gadgets (familiar action gadgets, such as "OK" or "Cancel").
- ☐ String gadgets (for text entry).
- ☐ Integer gadgets (for numeric entry).
- ☐ Checkboxes (for on/off items).
- ☐ Mutually exclusive gadgets (radio buttons).
- ☐ Cycle (multiple-choice toggle) gadgets.
- ☐ Sliders (to indicate a level).
- ☐ Scrollers (to indicate a position in a list or area).
- ☐ Listviews (scrolling lists of text).
- ☐ Palette (color selection) gadgets.
- ☐ Text-display (read-only) gadgets.
- ☐ Numeric-display (read-only) gadgets.

A. Manipulating GadTools Gadgets

1. Creating Gadgets

It is quite easy to create a gadget. Let us have a look at the code segment that would create the color slider we discussed earlier:

```
slidergad = CreateGadget( SLIDER_KIND, newgadget, prevgad,  
    GTSL_Min, 0,  
    GTSL_Max, 15,  
    GTSL_Level, 12,  
    TAG_DONE );
```

CreateGadget() typically allocates and initializes all the necessary Intuition structures, including the Gadget, IntuiText, and PropInfo structures, as well as certain buffers.

CreateGadget() takes a parameter that determines which of the above kinds of gadget you desire, a pointer to a GadTools NewGadget structure (which contains size and labelling information, among other things), and a pointer to the previously created gadget, for easy linking of gadgets. The list of tags follows these three parameters.

Since CreateGadget() is a tag-based function, it is easy to add more tags to get a fancier gadget. For example, GadTools can optionally display the running level beside the slider. The caller must supply an sprintf()-style formatting string, and the maximum length that string will resolve to when the number is inserted.

```
slidergad = CreateGadget( SLIDER_KIND, newgadget, prevgad,  
    GTSL_Min, 0,  
    GTSL_Max, 15,  
    GTSL_Level, 12,  
    GTSL_LevelFormat, "%ld"  
    GTSL_MaxLevelLen, 2,  
    TAG_DONE );
```

The level (0 to 15) would then be displayed beside the slider. The formatting string could instead be "%ld/15", so the level would be displayed as "0/15" through "15/15", if you wished.

2. Modifying Gadgets

Some of a gadget's attributes may be changed at any time, while others may only be specified at the time the gadget is created. For example, you may not change the level-formatting string after the gadget is created. However, you may change the slider's level (say to 5), and it is easy:

```
GT_SetGadgetAttrs( slidergad, win, req,  
    GTSL_Level, 5,  
    TAG_DONE );
```

Tags that may only be sent to CreateGadget() and not to GT_SetGadgetAttrs() will be marked as "create only". Those that are valid parameters to both functions will be marked as "create and set".

3. Handling Gadget Messages

When Intuition sends you a message for one of the gadgets, you should get and reply that message through a pair of special GadTools functions, `GT_GetIMsg()` and `GT_ReplyIMsg()`. These functions ensure that you see only the gadget events that concern you, and in a desirable form. For sliders, a message only gets through when the slider's level actually changes, and that level can be found in the `IntuiMessage`'s `Code` field as in:

```
imsg = GT_GetIMsg( win->UserPort );
object = imsg->IAddress;
class = imsg->Class;
code = imsg->Code;
GT_ReplyIMsg( imsg );
switch ( class )
{
    case MOUSEMOVE:
        if ( object == slidergad )
        {
            printf( "Slider at level %ld\n", code );
        }
        ...
        break;
    ...
}
```

4. IDCMP Flags

The various GadTools gadgets require certain classes of IDCMP messages in order to work. Each kind of gadget requires some of `GADGETUP`, `GADGETDOWN`, `MOUSEMOVE`, `MOUSEBUTTONS`, and `INTUITICKS`. You will find IDCMP definitions for each kind of gadget in `<libraries/gadtools.h/i>`. For example, `SLIDERIDCMP` is defined to be

```
(GADGETUP | GADGETDOWN | MOUSEMOVE)
```

Be sure to 'or' together the IDCMP definitions for all the kinds of gadgets you use (do not add them!). As well, even if you have no rendering of your own to do, you may not use `NOCAREREFRESH` window flag, and you must set the `REFRESHWINDOW` IDCMP flag.

B. The NewGadget Structure

The `NewGadget` structure is a set of information that typically is needed for most kinds of gadgets. Attributes that are common to the different gadgets have been placed in the `NewGadget` structure, while those that are unique to specific kinds of gadget are specified as tags sent to `CreateGadget()`. The `NewGadget` structure is defined as

```
struct NewGadget
{
    WORD    ng_LeftEdge, ng_TopEdge;
    WORD    ng_Width, ng_Height;
    UBYTE *ng_GadgetText;
    struct TextAttr *ng_TextAttr;
    UWORD   ng_GadgetID;
    ULONG   ng_Flags;
    APTR    ng_VisualInfo;
    APTR    ng_UserData;
};
```

Fields `ng_LeftEdge`, `ng_TopEdge`, `ng_Width`, and `ng_Height` define the size and position of the gadget you wish to create. Most gadgets have an associated label, which might be the text in a button or beside a checkmark. Point `ng_GadgetText` at the appropriate string. (Note that only the pointer to your text is copied, the text itself is not. The string you supply must remain constant and valid for the life of the gadget). You also need to specify an openable font (`ng_TextAttr`) to use for this label and other text that may be associated with the gadget. The `ng_Flags` field is used to describe general aspects of the gadget, which include where the label is to be placed (on the left side, the right side, centered above, centered below, or dead-center on the gadget), and whether the label should be rendered in the highlight color. For most gadget kinds, the label is placed on the left by default. Exceptions will be noted.

For your own use and convenience `ng_GadgetID` and `ng_UserData` are copied into the resulting gadget structure.

The `ng_VisualInfo` field must be set to a special GadTools structure (the `VisualInfo` structure) that contains information that is needed to create and render GadTools gadgets. There is a pair of GadTools functions to get and free the `VisualInfo` pointer. The `VisualInfo` structure itself is private to GadTools and subject to change. For this reason, its contents will not be documented.

C. The Kinds of GadTools Gadgets

1. Button Gadgets

Button gadgets (`BUTTON_KIND`) are perhaps the simplest and most straightforward kind of GadTools gadget. A button gadget would be used for an "OK" or "Save" or a similar action. You will get a hit-select button with a raised bevelled border, and the label you supply will be centered on the button's face. Since the label is not clipped, be sure that the gadget is wide enough to contain the text you supply.

Button gadgets recognize only one tag:

- ☐ `GA_DISABLED (BOOL)` - Set this attribute to `TRUE` to disable (ghost) the button gadget, to `FALSE` otherwise (defaults to `FALSE`). (May be done at `CreateGadget()` time or any time, using `GT_SetGadgetAttrs()`.)

When the user selects a button gadget, your program will receive a `GADGETUP` event.

If clicking on a button causes a requester to appear, for example a button that brings up a color requester or a "Quit" button that raises an "Are you sure?" requester, then the button text should end in "...", as in "Quit..."

2. Text-Entry and Number-Entry Gadgets

Text-entry (`STRING_KIND`) and number-entry (`INTEGER_KIND`) gadgets are fairly typical Intuition string gadgets. The typing area is contained by a border which is a raised ridge.

Text-entry gadgets accept the following tags:

- ☐ `GTST_String (STRPTR)` - A pointer to the string to be placed into the string gadget buffer, or `NULL` to get an empty string gadget. The default is `NULL`. The string itself is actually copied into the gadget's buffer. (This attribute may be set at `CreateGadget()` time, or later by calling `GT_SetGadgetAttrs()`.)
- ☐ `GTST_MaxChars (UWORD)` - The maximum number of characters that the string gadget should hold (default of 64). The string buffer that gets created for you will actually be one bigger than this number, in order to hold the trailing `NULL`. This attribute may only be set at `CreateGadget()` time.

- ❑ **STRINGA_Justification** - This attribute controls the placement of the string within its box, and can be one of **STRINGLEFT**, **STRINGRIGHT**, or **STRINGCENTER**. The default is **STRINGLEFT**. (Create only.)
- ❑ **STRINGA_ReplaceMode** (BOOL) - Set **STRINGA_ReplaceMode** to **TRUE** to get a string gadget which is in replace-mode, as opposed to auto-insert mode. (Create only.)
- ❑ **GA_DISABLED** (BOOL) - Set this attribute to **TRUE** to disable the string gadget, to **FALSE** otherwise (defaults to **FALSE**). (Create or set.)

Number-entry gadgets accept the following tags:

- ❑ **GTIN_Number** (ULONG) - The number to be placed into the integer gadget. The default is zero. (Create or set.)
- ❑ **GTIN_MaxChars** (UWORD) - The maximum number of digits that the integer gadget should hold (defaults to 10). The string buffer that gets created for you will actually be one bigger than this, in order to hold the trailing **NULL**. (Create only.)
- ❑ **GA_DISABLED** (BOOL) - Set this attribute to **TRUE** to disable the integer gadget, to **FALSE** otherwise (defaults to **FALSE**). (Create or set.)

As with all Intuition string gadgets, you will receive a **GADGETUP** message only when the user presses **<ENTER>** while typing in the gadget. Note that also like Intuition string gadgets, you do not hear anything if the user deactivates the string gadget by clicking elsewhere. Therefore it is a good idea to always check the string gadget's buffer before you use its contents, instead of just tracking its contents as you hear **GADGETUP** messages for this gadget.

To read the string gadget's buffer, look at the Gadget's **StringInfo** Buffer:

```
( ( struct StringInfo * )gad->SpecialInfo )->Buffer
```

To determine the value of an integer gadget, look at the Gadget's **StringInfo** **LongInt** in the same way.

Of course, you should always use the **GTST_String** or **GTIN_Number** tags to set these values, and never write to the **StringInfo->Buffer** or **StringInfo->LongInt** fields directly.

3. Checkboxes

Checkboxes (**CHECKBOX_KIND**) are appropriate whenever you need to present an option which may be turned on or off. This kind of gadget consists of a raised box which contains a checkmark if the option is selected, or is blank if the option is not selected. Clicking on the box toggles the state of the checkbox.

The dimensions of a checkbox are currently fixed. If variable-sized checkboxes are added in the future, they will be done in a compatible manner. Currently the width and height specified in the **NewGadget** structure are ignored in favor of the fixed width and height. You may control the checkbox with the following tags:

- ❑ **GTCB_Checked** (BOOL) - Set this attribute to **TRUE** to set the gadget's state to "checked", or set it to **FALSE** to mark the gadget as "unchecked" (defaults to **FALSE**). (Create or set.)
- ❑ **GA_DISABLED** (BOOL) - Set this attribute to **TRUE** to disable the checkbox, to **FALSE** otherwise (defaults to **FALSE**). (Create or set.)

You will receive an `IntuiMessage` with a class of `GADGETUP` whenever the user selects a checkbox. As this gadget always toggles, you can easily track the state of the gadget yourself. Otherwise, feel free to look at the `gadget->Flags` `SELECTED` bit. Of course, the gadget structure itself is not synchronized to the `IntuiMessages` you receive. If the user quickly clicks a second time, the `SELECTED` bit can toggle again before you get a chance to read it. This is true of any of the dynamic fields of the gadget structure. It is worth being aware of this, although only rarely will you have to account for it in your code.

4. Mutually-Exclusive Gadgets

You should use mutually exclusive gadgets (`MX_KIND`), or "radio buttons", when the user must choose one option from a short list of possibilities. A set of mutually exclusive gadgets consists of a list of labels, and beside each label is a small raised dot. Exactly one of the dots is recessed and highlighted, to indicate the selected choice. The user can pick another choice by clicking on any of the raised dots. This choice will become active, and the previously selected choice will become inactive. That is, the selected dot will become recessed while the previous one will pop out, like the buttons on a car radio. Mutually exclusive gadgets are appropriate when there are a small number of choices, perhaps eight or less.

Mutually exclusive gadgets recognize these tags:

- ☐ `GTMX_Labels (STRPTR *)` - A NULL-pointer-terminated array of strings which are to be the labels beside each choice in the set of mutually exclusive gadgets. This array determines how many buttons are created. This array must be supplied to `CreateGadget()`, and may not be changed. The strings themselves must remain valid for the lifetime of the gadget. (Create only.)
- ☐ `GTMX_Active (UWORD)` - The ordinal number (counting from zero) of the active choice of the set of mutually exclusive gadgets (defaults to zero). (Create or set.)
- ☐ `GTMX_Spacing (UWORD)` - The amount of space (rows of pixels) that will be placed between successive choices in a set of mutually exclusive gadgets (defaults to 1). (Create only.)

When the user selects a new choice from a set of mutually exclusive gadgets, you will receive a `GADGETDOWN` `IntuiMessage`. You may look in the `IntuiMessage`'s `Code` field for the ordinal number of the new active selection. Currently, mutually exclusive gadgets may not be disabled.

The `ng_GadgetText` field of the `NewGadget` structure is ignored for mutually exclusive gadgets. The text position specified in `ng_Flags` determines whether the item labels are placed to the left or the right of the radio buttons themselves. By default, the labels appear on the left. Do not specify `PLACETEXT_ABOVE`, `PLACETEXT_BELOW`, or `PLACETEXT_IN` for this kind of gadget.

Like the checkbox, the size of the radio button is currently fixed, and the dimensions you supply in the `NewGadget` structure are ignored. If in the future the button glyph is made scalable, it will be done in a compatible manner.

5. Cycle Gadgets

Like mutually exclusive gadgets, cycle gadgets (`CYCLE_KIND`) allow the user to choose one option from among several. The cycle gadget appears as a raised rectangular button with a vertical divider near the left side. A circular arrow glyph appears to the left of the divider, while the current choice appears to the right. Clicking on the cycle gadget advances to the next choice, while shift-clicking on it changes it to the previous choice. Cycle gadgets are more compact than mutually exclusive gadgets, since only the current choice is displayed. They are preferable to mutually exclusive gadgets when a window needs to have several such gadgets (for example, the `PrinterGfx` Preferences), or when there is a medium number of choices. If the number of choices is much more than about a dozen, it may become too frustrating and inefficient for the user to find the desired choice. In that case, give the user a listview (scrolling list) instead.

The tags recognized by cycle gadgets are:

- ❑ **GTCY_Labels (STRPTR *)** - Like **GTMX_Labels**, this tag is associated with a NULL- pointer-terminated array of strings which are the choices that this gadget allows. This array must be supplied to **CreateGadget()**, and may not be changed. The strings themselves must remain valid for the lifetime of the gadget. (Create only.)
- ❑ **GTCY_Active (UWORD)** - The ordinal number (counting from zero) of the active choice of the cycle gadget (defaults to zero). (Create or set.)

When the user clicks or shift-clicks on a cycle gadget, you will receive a **GADGETUP IntuiMessage**. You may look in the **Code** field of the **IntuiMessage** for the ordinal number of the new active selection. Currently, cycle gadgets may not be disabled.

6. Sliders

Sliders are one of the two kinds of proportional gadgets offered by **GadTools**. Slider gadgets (**SLIDER_KIND**) are used to control an amount, a level, or an intensity, such as volume or color. Optionally, the current level of the slider may be displayed (in real-time) alongside the gadget. Slider gadgets accept the following tags:

- ❑ **GTSL_Min (WORD)** - The minimum level of a slider (defaults to 0). (Create or set.)
- ❑ **GTSL_Max (WORD)** - The maximum level of a slider (defaults to 15). (Create or set.)
- ❑ **GTSL_Level (WORD)** - The current level of a slider (defaults to 0). When the level is at its minimum, the knob will be all the way to the left (for a horizontal slider) or all the way at the bottom (for a vertical slider). Conversely, the maximum level corresponds to the knob being to the extreme right or top. (Create or set.)
- ❑ **GTSL_LevelFormat (STRPTR)** - A C-style formatting string used to render the slider level beside the slider. Be sure to use the 'l' (long) modifier for the number. The simplest would be "%ld". You could make a 2-digit hexadecimal slider with "%02lx". Things like "%ld hours" are permissible. See the autodocs for **exec/RawDoFmt()** for full details. By default, the level is not displayed. (Create only, but if you specify this tag, you must also provide **GTSL_MaxLevelLen**.)
- ❑ **GTSL_MaxLevelLen (UWORD)** - The maximum length of the string that will result from your level-formatting string. By default, the level is not displayed. (Create only, but if you specify this tag, you must also provide **GTSL_LevelFormat**.)
- ❑ **GTSL_LevelPlace** - To choose where the optional display of the level is positioned. It must be one of **PLACETEXT_LEFT**, **PLACETEXT_RIGHT**, **PLACETEXT_ABOVE**, or **PLACETEXT_BELOW** (defaults to left). You may place the level anywhere (same or different as the gadget label's place) with the following exception: the level and the label may not be both above or both below the gadget. If you want them on the same side, allow space in the gadget's label (see the example). (Create only.)
- ❑ **GTSL_DisFunc (LONG (*function)(struct Gadget *, WORD))** - Optional function to convert the level for display. A slider to select the number of colors for a screen may operate in screen depth (**GTSL_Min** = 1, **GTSL_Max** = 5), but actually display the number of colors (2, 4, 8, 16, or 32) by providing a **GTSL_DisFunc** function that returns (1 << level). Your function must take a pointer to the gadget as the first parameter and the level (a **WORD**) as the second, and return the result as a **LONG**. If you have requested that the level be displayed, then by default the level is displayed without any conversion. (Create only.)

- ☐ **GA_IMMEDIATE (BOOL)** - Set this to TRUE to receive a GADGETDOWN IntuiMessage when the user presses the mouse button over the slider (defaults to FALSE). (Create only.)
- ☐ **GA_RELVERIFY (BOOL)** - Set this to TRUE to receive a GADGETUP IntuiMessage when the user releases the mouse button after using the slider (defaults to FALSE). (Create only.)
- ☐ **PGA_FREEDOM** - Set to LORIENT_VERT for a vertical slider or LORIENT_HORIZ for a horizontal slider (defaults to horizontal). (Create only.)
- ☐ **GA_DISABLED (BOOL)** - Set this attribute to TRUE to disable the slider, to FALSE otherwise (defaults to FALSE). (Create or set.)

You may receive up to three different kinds of IntuiMessage when the user plays with a slider, those of Class MOUSEMOVE, GADGETUP, and GADGETDOWN. You may examine the IntuiMessage Code field to discover the slider's level. You will hear MOUSEMOVE IntuiMessages whenever the slider's level changes. You never hear MOUSEMOVE IntuiMessages when the knob has not moved far enough for the level to actually change. For example if your slider runs from 0 to 15 and is currently set to 12, if the user drags the slider all the way up you will hear no more than three MOUSEMOVEs, one each for 13, 14, and 15. If you have set {GA_IMMEDIATE, TRUE}, then you will always hear a GADGETDOWN IntuiMessage when the user begins to adjust a slider. If you have set {GA_RELVERIFY, TRUE}, then you will always hear a GADGETUP IntuiMessage when the user finishes adjusting the slider. If you have asked for GADGETUP and/or GADGETDOWN IntuiMessages, you will always hear them, even if the level has not changed since the previous IntuiMessage.

Note that the Code field of the IntuiMessage structure is a UWORD, while the slider's level may be negative, since it is a WORD. Be sure to copy or cast the IntuiMessage->Code into a WORD if your slider has negative levels.

If the user clicks in the container next to the knob, the slider level will increase or decrease by one. If the user drags the knob itself, then the knob will snap to the nearest integral position when it is released. Here is an example of the screen-depth slider discussed earlier:

```

/* NewGadget initialized here. Note the three spaces
   after "Slider:", to allow a blank plus the two digits
   of the level display */
ng.ng_Flags = PLACETEXT_LEFT;
ng.ng_GadgetText = "Slider:  ";

gad = CreateGadget( SLIDER_KIND, gad, &ng,
    GTSL_Min, 1,
    GTSL_Max, 5,
    GTSL_Level, current_depth,
    GTSL_MaxLevelLen, 2,
    GTSL_LevelFormat, "%2ld",
    GTSL_DisFunc, DepthToColors,
    TAG_DONE );

...
LONG DepthToColors( gad, level )
    struct Gadget *gad;
    WORD level;
{
    return ( (WORD) ( 1 << level ) );
}

```

7. Scrollers

Scrollers (`SCROLLER_KIND`) bear some similarity to sliders, but are used for a quite different job: they allow the user to adjust the position of a view of a larger area. For example, Workbench's windows have scrollers that allow you to pan about and see icons that are outside the visible portion of a window. A scrolling list in a file requester has a scroller that allows you to see different parts of the whole list. A scroller consists of a proportional gadget, and usually also has a pair of arrow buttons.

While the slider deals in minimum, maximum, and current level, the scroller understands `Total`, `Visible`, and `Top`. For a scrolling list, `Total` would be the number of items in the whole list, while `Visible` would be the number of lines visible in the list and `Top` would be the first line shown in the visible part of the list. `Top` would run from zero to `(Total-Visible)`. For an area-scroller such as those in Workbench's windows, `Total` would be the height (or width) of the whole area, `Visible` would be the visible height (or width), and `Top` would be the top (or left) edge of the visible part.

Scrollers respect the following tags:

- ☐ `GTSC_Top (WORD)` - The top line or position visible in the area that the scroller represents (defaults to 0). (Create or set.)
- ☐ `GTSC_Total (WORD)` - The total size of the area that the scroller represents (defaults to 0). (Create or set.)
- ☐ `GTSC_Visible (WORD)` - The visible size of the area that the scroller represents (defaults to 2). (Create or set.)
- ☐ `GTSC_Arrows (UWORD)` - Asks for arrow gadgets to be attached to the scroller. The value supplied will be used as the width of each arrow button for a horizontal scroller, or the height of each arrow button for a vertical scroller (the other dimension will match the whole scroller). By default, no arrows will be attached, though we generally recommend that you ask for arrows. (Create only.)
- ☐ `GA_IMMEDIATE (BOOL)` - Set this to `TRUE` to receive a `GADGETDOWN` IntuiMessage when the user presses the mouse button over the scroller (defaults to `FALSE`). (Create only.)
- ☐ `GA_RELVERIFY (BOOL)` - Set this to `TRUE` to receive a `GADGETUP` IntuiMessage when the user releases the mouse button after using the scroller (defaults to `FALSE`). (Create only.)
- ☐ `PGA_FREEDOM` - Set to `LORIENT_VERT` for a vertical scroller or `LORIENT_HORIZ` for a horizontal scroller (defaults to horizontal). (Create only.) `GA_DISABLED (BOOL)` - Set this attribute to `TRUE` to disable the scroller, to `FALSE` otherwise (defaults to `FALSE`). (Create or set.)

The IntuiMessages you receive for a scroller gadget are the same in nature as those for a slider (see above), including the fact that you only hear a message when the knob moves far enough for the `Top` value to actually change. The `Code` field of the IntuiMessage will contain the new `Top` value of the scroller. If the user clicks on an arrow gadget, the scroller moves by one unit. If the user holds the button down over an arrow gadget, it repeats.

If the user clicks in the container next to the knob, the scroller will move by one page, which is the visible amount less one. This means that when the user pages through a scrolling list, any pair of successive views will overlap by one line. This helps the user understand the continuity of the list. If you are using a scroller to pan through an area then there will be an overlap of one unit between successive views. We recommend that you scale your `Top`, `Visible`, and `Total` so that one unit represents about five to ten percent of the visible amount. A future GadTools might allow you to explicitly set the amount of overlap.

8. Listview Gadgets

Listview gadgets (LISTVIEW_KIND) are scrolling lists. They consist of a scroller with arrows, an area where the list itself is visible, and optionally a place where the current selection is displayed, which may be editable. The user can browse through the list using the scroller or its arrows, and may select an entry by clicking on that item. There are a number of tags that are used with listviews:

- ☐ GTLV_Labels (struct List *) - An Exec list whose nodes' In_Name fields are to be displayed as items in the scrolling list. If your list is empty, you can use an empty List structure or a NULL value for GTLV_Labels. Use a value of (~0) to detach the list from the listview (see below). Defaults to NULL. (Create or set.)
- ☐ GTLV_Top (UWORD) - The top item in the list visible in the listview (defaults to 0). (Create or set.)
- ☐ GTLV_ReadOnly (BOOL) - Set this to TRUE for a read-only listview, which the user can browse, but not select items from (defaults to FALSE). A read-only listview can be recognized because the list area is recessed, not raised. (Create only.)
- ☐ GTLV_ScrollWidth (UWORD) - The width of the scroller to be used in the listview (defaults to 16). (Create only.)
- ☐ GTLV_ShowSelected (struct Gadget *) - If you want the currently selected entry displayed underneath the listview, then use this tag. Set its value to NULL to get a read-only (TEXT_KIND) display of the currently selected entry, or set it to a pointer to an already- created GadTools string gadget, to allow the user to directly edit the current entry. By default, there will be no display of the currently selected entry. (Create only.)
- ☐ GTLV_Selected (UWORD) - Ordinal number of the item to be placed into the display of the current selection under the listview. This tag is ignored if GTLV_ShowSelected is not used. Set it to (~0) to have no current selection (defaults to ~0). (Create or set.)
- ☐ LAYOUTA_SPACING (UWORD) - Extra space (rows of pixels) to be placed between the entries in the listview (defaults to zero). (Create only.)

The only IntuiMessages you ever hear from a listview happen when the user selects an item from the list. You will then receive a GADGETUP IntuiMessage, and you may look at the Code field of that message for the ordinal number of the item within the list that was selected. This number is independent which part of the list the user has scrolled into view.

If you attach a display gadget by using the TagItem {GTLV_ShowSelected, NULL}, then whenever the user clicks on an entry in the listview it will be copied into the display gadget underneath. If you want this to be editable, then you must first create a GadTools string gadget whose width matches the width of the listview, and then use the TagItem {GTLV_ShowSelected, stringgad}, where stringgad is a pointer to that gadget. When the user selects any entry from the listview, it gets copied into the string gadget. As well, the user can edit the string, and you will hear normal string gadget GADGETUP messages when the user presses <ENTER>.

The Exec list and its node structures may not be modified while they are attached to the listview, since the list might be needed at any time. If you have prepared a whole new List structure, you may replace the displayed list in a single step by calling GT_SetGadgetAttr() with the TagItem {GTLV_Labels, newlist}. If you need to operate on the list that you have already passed to the listview, detach it by setting the GTLV_Labels attribute to (~0). When you are done modifying the list, resubmit it by setting GTLV_Labels to once again point to it. This is better than first setting the labels to NULL and later back to your list, since setting GTLV_Labels to NULL will clear the listview. If you do set the GTLV_Labels attribute to (~0), you are expected to set it back to something determinate (a list, or NULL) soon after.

The height you specify for your listview will determine the number of lines in the list area. When you create a listview, it will be no bigger than the size you specify in the NewGadget structure. The size will include the current-display gadget (if any) that you have requested via the GTLV_ShowSelected tag. The listview may end up being less tall than you asked for, since its allowable height is as granular as the height of a single line in the list area.

By default, the gadget label will be placed above the listview. You may override this using ng_Flags. A listview may not be disabled.

9. Palette Gadgets

Palette gadgets (PALETTE_KIND) let the user pick a color from a set of several. A palette gadget consists of a number of colored squares, one for each color available. As well, an indicator square which is filled with the currently selected color is optional. To create a color editor, a palette gadget would be combined with some sliders to control red, green, and blue components, for example.

Palette gadgets use the following tags:

- ☐ GTPA_Depth (UWORD) - The number of bitplanes that the palette represents. There will be $(1 \ll \text{depth})$ squares in the palette gadget. The default depth is 1. (Create only.)
- ☐ GTPA_Color (UBYTE) - The selected color of the palette (defaults to 1). (Create or set.)
- ☐ GTPA_ColorOffset (UBYTE) - The first color to use in the palette. If GTPA_Depth was two and GTPA_ColorOffset was four, then the palette would have squares for colors four, five, six, and seven. (Defaults to zero). (Create only.)
- ☐ GTPA_IndicatorWidth (UWORD) - The desired width of the current-color indicator. By specifying this tag, you are asking for an indicator to be placed to the left of the color selection squares. The indicator will be as tall as the gadget itself. By default there is no indicator. (Create only.)
- ☐ GTPA_IndicatorHeight (UWORD) - The desired height of the current-color indicator. By specifying this tag, you are asking for an indicator to be placed above the color selection squares. The indicator will be as wide as the gadget itself. By default there is no indicator. (Create only.)
- ☐ GA_DISABLED (BOOL) - Set this attribute to TRUE to disable the palette gadget, to FALSE otherwise (defaults to FALSE). (Create or set.)

You will receive a GADGETUP IntuiMessage when the user selects a color from the palette. The current-color indicator is recessed, indicating that clicking on it has no effect.

If your palette is wide and not tall, you should use the GTPA_IndicatorWidth tag to put the indicator on the left. If your palette is tall and narrow, put the indicator on top using GTPA_IndicatorHeight. By default, the gadget's label will go above the palette gadget, unless you specify GTPA_IndicatorWidth, in which case the label will go on the left. In either case, you may override the default by setting the appropriate flag in the NewGadget's ng_Flags field.

The size you specify for your palette gadget will determine how the area is subdivided to make the individual color squares. The actual size of the palette gadget will be no bigger than the size you supply, but it can be smaller since the color squares will end up all exactly the same size.

10. Text-Display and Number-Display Gadgets

Text-display (TEXT_KIND) and number-display (NUMBER_KIND) gadgets are read-only displays of information. They are useful for displaying information that is not editable or selectable, but that you nevertheless want to use some of the GadTools formatting and visuals for. Conveniently, they are automatically refreshed for you, like all GadTools gadgets. As well, their displayed value can be changed by your program.

These gadgets can consist of the label supplied as the NewGadget's `ng_GadgetText` and/or a fixed or changing numeric value or string.

Text-display gadgets recognize the following tags:

- ☐ `GTTX_Text (STRPTR)` - Pointer to the string to be displayed, or NULL for no string (defaults to NULL). (Create or set.)
- ☐ `GTTX_Border (BOOL)` - Set to TRUE to place a recessed border around the displayed string (defaults to FALSE). (Create only.)
- ☐ `GTTX_CopyText (BOOL)` - This flag instructs the text-display gadget to copy the supplied `GTTX_Text` string, instead of using only a pointer to the string. This only works for the initial value of `GTTX_Text` set at `CreateGadget()` time. If you subsequently change `GTTX_Text`, the new text will be referenced by pointer, not copied. (Create only.)

Number-display gadgets have the following tags:

- ☐ `GTNM_Number (LONG)` - The number to be displayed (defaults to zero). (Create or set.)
- ☐ `GTNM_Border (BOOL)` - Set to TRUE to place a recessed border around the displayed number (defaults to FALSE). (Create only.)

Text-display and number-display gadgets never cause `IntuiMessages` to be sent, since they are not selectable.

11. Generic Gadgets

If you need to define your own gadgets, but you like the convenience of GadTools gadget creation and deletion, you may create a GadTools generic gadget and use it any way you see fit. In fact, all of the kinds of GadTools gadgets are created out of GadTools `GENERIC_KIND` gadgets.

The gadget that gets created will heed almost all the information contained in the `NewGadget` structure you supply. If you supply a non-NULL `ng_GadgetText`, the gadget's `GadgetText` will point to an `IntuiText` structure with the supplied string and font. However, do not specify any of the `PLACETEXT` `ng_Flags`, as they are currently ignored by `GENERIC_KIND` gadgets, but this may not always be so.

It is up to you to set the `Flags`, `Activation`, `GadgetRender`, `SelectRender`, `MutualExclude` and `SpecialInfo` fields of your gadget structure. As well, you must set the `GadgetType` field, but be certain to preserve the bits set in it as it comes back from `CreateGadget()` - they are non-zero. If you wanted to make a boolean gadget, you would therefore say:

```
gad->GadgetType |= BOOLGADGET;  
and not  
gad->GadgetType = BOOLGADGET;
```

(If you had just used "=", the gadget would not get freed by `FreeGadgets()`.)

D. Functions for GadTools Gadgets

GadTools provides a number of functions to create, modify, handle, refresh, and free gadgets. Some of them are used to do new kinds of work, such as `CreateGadget()` and `GT_SetGadgetAttrs()`, while others should be used in place of similar Intuition functions, to cooperate with GadTools.

1. `CreateGadget()`

As its name implies, `CreateGadget()` (and the tag-array version, `CreateGadgetA()`) is used to allocate and initialize new GadTools gadgets. `CreateGadget()` takes three parameters, followed by a set of tags. The first parameter indicates which kind of gadget you wish to create. You may choose from among the kinds already discussed.

The second parameter is a pointer to the previous gadget you have created. This has three useful effects. First, the new gadget is automatically linked into the previous gadget's `NextGadget` field. Second, if one of the gadget creations fails (usually due to low memory, but other causes are possible), then for the next call to `CreateGadget()`, `previous` will be `NULL`, and `CreateGadget()` will fail instantly. This means that you can perform several successive calls to `CreateGadget()`, and only have to check for failure at the end. The third reason is that (although you never need to know this) certain calls to `CreateGadget()` actually cause several Intuition gadgets to be allocated, and they can be automatically linked in without your interaction only if a previous gadget pointer is supplied (see "Documented Side-Effects" for a warning). If several gadgets are created, they work together to give you the functionality that is a single GadTools "gadget", and you should always act as though the gadget pointer you receive is the one true gadget.

There is one exception to the fact that you only have to check for failure after the last `CreateGadget()` call, and that is when you are depending on the successful creation of a gadget. Say you wanted to create a string gadget and save a pointer to the string buffer. You should do as follows:

```
gad = CreateGadget( STRING_KIND, gad, &ng,
    GTST_String, "Hello World", TAG_DONE );
if ( gad )
{
    stringbuffer = ( ( struct StringInfo * )
        ( gad->SpecialInfo ) )->Buffer;
}
/* Creation can continue here: */
gad = CreateGadget( ..._KIND, gad, &ng2, ... );
```

The third parameter to `CreateGadget()` is a pointer to the `NewGadget` structure, as has been explained earlier, and can be found in `<libraries/gadtools.h>`. One major benefit of having a reusable `NewGadget` structure is that often many fields do not change, and some fields change incrementally. For example, you typically can set the `NewGadget`'s `ng_VisualInfo` and `ng_TextAttr` only once, and never have to modify them again. A set of similar gadgets may share size and some positional information. You may end up with something like:

```
/* Say that the NewGadget structure 'ng' is fully
   initialized here for a button labelled "OK" */

gad = CreateGadget( BUTTON_KIND, gad, &ng, TAG_DONE );

/* Modify only those fields that have changed: */
ng.ng_GadgetID++;
ng.ng_LeftEdge += 80;
ng.ng_GadgetText = "Cancel";
gad = CreateGadget( BUTTON_KIND, gad, &ng, TAG_DONE );
```

After the first three parameters comes the tags. As explained earlier, `CreateGadgetA()` takes a pointer to `TAG_DONE`-terminated array of `TagItems`, while `CreateGadget()` expects to find a set of `TagItems` (ending with `TAG_DONE`) on the caller's stack. The tags that are valid depend on the kind of gadget being created. The individual tags have been explained in the section on each kind of gadget, and may also be found in the autodocs for `CreateGadget()`. Only those tags which are noted as being valid at "create" time may be used. Creating certain kinds of gadgets may fail in the absence of some required tags.

All gadgets created by GadTools currently have the `GADTOOL_TYPE` bit set in their `GadgetType` field. It is not correct to depend on this or otherwise assume this will remain true.

2. `GT_SetGadgetAttrs()`

To change the attributes of a GadTools gadget after it has been created, you should call the `GT_SetGadgetAttrs()` function. This function takes a pointer to a GadTools gadget (as returned by `CreateGadget()`) as its first parameter, followed by a pointer to the gadget's window. The third parameter for `GT_SetGadgetAttrs()` is currently required to be `NULL`, but it is reserved as a pointer to a requester, in anticipation of a future GadTools that is allowed to be used in requesters.

Following these three parameters, you pass a set of `TagItems` on the stack (for `GT_SetGadgetAttrs()`) or a pointer to an array of `TagItems` (for `GT_SetGadgetAttrsA()`) that describe the attributes you would like to change. As usual, the last `TagItem` should be `TAG_DONE`. In the sections describing the kinds of gadgets in GadTools, the possible tags are described. Only those marked as valid at "set" time may be changed with `GT_SetGadgetAttrs()`. The tags are also described in the autodocs for `GT_SetGadgetAttrs()`.

When you change a gadget using this function, the gadget will automatically update its visuals. No refresh is required, nor should any refresh call be performed. Note that you may not call `GT_SetGadgetAttrs()` inside of `GT_Begin/EndRefresh()`, which is true of other Intuition gadget functions as well.

Here are some example uses of `GT_SetGadgetAttrs()`:

```
/* Disable a button gadget */
GT_SetGadgetAttrs( buttongad, win, NULL,
    GA_DISABLED, TRUE,
    TAG_DONE );

/* Change a slider's range to be 1 to 100, currently at 50 */
GT_SetGadgetAttrs( slidergad, win, NULL,
    GTSL_Min, 1,
    GTSL_Max, 100,
    GTSL_Level, 50,
    TAG_DONE );

/* Add a node to the head of listview's list,
   and make it the selected one */
GT_SetGadgetAttrs( listviewgad, win, NULL,
    GTLV_Labels, ~0, /* detach list before modifying */
    TAG_DONE );
AddHead( &lvlabels, &newnode );
GT_SetGadgetAttrs( listviewgad, win, NULL,
    GTLV_Labels, &lvlabels, /* re-attach list */
    GTLV_Selected, 0,
    TAG_DONE );
```

3. *GetVisualInfo()* and *FreeVisualInfo()*

In order to ensure their best appearance, GadTools gadgets (and menus - see later) need to know various pieces of information about the screen they will appear on. Before creating any GadTools gadgets or menus, you must get this information using the *GetVisualInfo()* call. The first parameter to *GetVisualInfo()* is a pointer to the screen you will use. *GetVisualInfo()* also accepts a set of tags, though currently none are recognized, so you should always use only *TAG_DONE*. The function returns an abstract handle called the *VisualInfo*. You must set the *ng_VisualInfo* field of your *NewGadget* structures to this handle. As well, certain GadTools menu and rendering functions require the *VisualInfo* handle.

There are several ways to get the pointer to the screen you will be opening your window on. If you are going to be using a custom screen, then you get this pointer by calling *OpenScreen()* or *OpenScreenTags()*. If you are opening your window on the default public screen (which is usually the Workbench), then the screen pointer is found in *window->WScreen*. However, usually you will want to create your gadgets and menus before you open your window. In that case, you should use the *Intuition LockPubScreen()* call to get a pointer to the default public screen, which also gets you a lock to prevent the screen from closing on you.

After all the gadgets and menus have been freed, but before you release the screen pointer (by calling *CloseScreen()*, *CloseWindow()*, or *UnlockPubScreen()*, depending on your technique), you must call *FreeVisualInfo()*, which takes the *VisualInfo* handle as its parameter.

The sequence of events would then look like this:

```
init()
{
    myscreen = LockPubScreen( NULL );
    if ( !myscreen )
    {
        cleanup( "Failed to lock default public screen" );
    }
    vi = GetVisualInfo( myscreen );
    if ( !vi )
    {
        cleanup( "Failed to GetVisualInfo" );
    }
    /* Create your gadgets here */
    ng.ng_VisualInfo = vi;
    ...
}

void cleanup(errorstr)
STRPTR errorstr;
{
    /* These functions may be safely called with a NULL parameter: */
    FreeGadgets( glist );
    FreeVisualInfo( vi );

    if ( myscreen )
        UnlockPubScreen( NULL, myscreen );

    printf(errorstr);
}
```

4. *GT_GetMsg()* and *GT_ReplyMsg()*

You may have noticed that the *IntuiMessages* you receive from GadTools contain more information (in the *Code* field) than you find in regular Intuition gadget messages, and that a lot of unnecessary messages (mostly *MOUSEMOVES*) never seem to be sent. This is one of the reasons that dealing with GadTools gadgets is much easier than dealing with regular Intuition gadgets. Unfortunately this sort of thing cannot happen magically, and it requires a small amount of cooperation on your part. This cooperation consists of using the GadTools functions *GT_GetMsg()* and *GT_ReplyMsg()* where you would normally have used the *Exec GetMsg()* and *ReplyMsg()* calls to manage your *IntuiMessages*.

GT_GetMsg() actually calls *GetMsg()* to remove a message from the specified port (your window's *UserPort*). If the message pertains to a GadTools gadget then some dispatching code in GadTools will be called to process the message. What you will receive from *GT_GetMsg()* is actually a copy of the real *IntuiMessage*, possibly with some supplementary information from GadTools, such as what you typically find in the *Code* field. *GT_ReplyMsg()* will take care of cleaning up and replying the real *IntuiMessage*. This description of the inner workings of *GT_GetMsg()* and *GT_ReplyMsg()* is provided for understanding only; it is crucial that you make no assumptions or interpretations about the real *IntuiMessage*. Any such inferences are very likely to not hold true in the future. See the section on documented side-effects for more discussion.

5. *GT_RefreshWindow()*

In the usual scheme of things, you create gadgets, add them to your window, and then call Intuition's *RefreshGList()* function to draw all the gadgets. To complete the rendering of GadTools gadgets, you are also required to call the *GT_RefreshWindow()* function. This function takes a pointer to your window, and a pointer to a requester as its parameters. As GadTools gadgets are not currently supported in requesters, the second parameter must currently be *NULL*.

6. *GT_BeginRefresh()* and *GT_EndRefresh()*

By the time you receive a *REFRESHWINDOW* IDCMP message for your window, Intuition has already refreshed its gadgets. You then call Intuition's *BeginRefresh()*, then do your custom rendering operations, and finally you call *EndRefresh()*. To allow the GadTools gadgets to be fully refreshed, you simply use *GT_BeginRefresh()* and *GT_EndRefresh()* in the same way. If you are using GadTools, you may not set your Window's *NOCAREREFRESH* Flag. Even if you have no custom rendering of your own, you must have this minimum code to handle *REFRESHWINDOW* IDCMP messages:

```
case REFRESHWINDOW:
    GT_BeginRefresh( win );
    /* your own custom rendering, if any, goes here */
    GT_EndRefresh( win, TRUE );
    break;
```

7. *FreeGadgets()*

After you have closed down your window, it will be time to free the gadgets that you allocated using *CreateGadget()*. *FreeGadgets()* is a simple call that will free all the GadTools gadgets that it finds, beginning with the gadget whose pointer you pass to it. Any non-GadTools gadgets found on the list will not be freed, but they will not necessarily form a nice list anymore, since any intervening GadTools gadgets will be gone. It is safe to call *FreeGadgets()* with a *NULL* gadget pointer.

8. *CreateContext()*

The Gadget Toolkit requires some per-window context information. *CreateContext()* establishes a place for that information to go. Before you create any Toolkit gadgets, you should call this function. *CreateContext()* takes a double-pointer to a Gadget structure as its parameter. More specifically, it wants a pointer to a NULL-initialized pointer to a Gadget structure. This pointer to the Gadget structure (glist in the example below) may then serve as a handle to the list of gadgets as they are created.

CreateContext() actually creates an invisible unselectable gadget that you need not worry about. GadTools can quickly find this gadget to locate the context information. The return value of *CreateContext()* is a pointer to this gadget, which should be fed to your first call to *CreateGadget()*.

Typically, your code would look like:

```
struct Gadget *glist = NULL;
struct Gadget *gad;

/* Note well that CreateContext() requires a POINTER to
   a NULL-initialized pointer to struct Gadget: */
gad = CreateContext( &glist );

/* Your gadget creation code goes here: */
gad = CreateGadget( BUTTON_KIND, gad, ... );
gad = CreateGadget( STRING_KIND, gad, ... );
gad = CreateGadget( MX_KIND, gad, ... );

if ( !gad )
{
    FreeGadgets( glist );
    exit_error();
}
else
{
    AddGList( win, glist, -1, -1, NULL );
    GT_RefreshWindow( win, NULL );
    /* and continue on... */
}
```

9. *GT_FilterIMsg()* and *GT_PostFilterIMsg()*

For most GadTools users, *GT_GetIMsg()* and *GT_ReplyIMsg()* work perfectly well. However, in rare cases you may find that they pose a bit of a problem. A typical case is when all your messages are supposed to go through a centralized *ReplyMsg()* that cannot be converted to a *GT_ReplyIMsg()*. Since calls to *GT_GetIMsg()* and *GT_ReplyIMsg()* must be paired, there would be a problem.

For such cases, the *GT_FilterIMsg()* and *GT_PostFilterIMsg()* functions are available. In such a program you would use Exec's *GetMsg()* as usual. When you need to invoke the Gadget Toolkit (for example after having determined that this message applies to one of the windows that contains GadTools gadgets), you should call *GT_FilterIMsg()*, which takes a regular *IntuiMessage* and returns either a cooked *IntuiMessage* or a NULL, signifying that that message was "consumed" by a GadTools gadget. Consumed messages are NOT replied for you.

If you do get a filtered message pointer from `GT_FilterIMsg()`, then you should use it like any message you would have got from `GT_GetIMsg()`. When you are done with it, you must call `GT_PostFilterIMsg()`. In all cases, you **MUST** then reply the original `IntuiMessage` using `ReplyMsg()`, or do whatever else you need before replying.

```
/* port is a message port receiving different messages */
/* gtwindow is your window that has GadTools gadgets */

imsg = GetMsg( port );

/* Is this the window with GadTools gadgets? */
if ( imsg->IDCMPWindow == gtwindow )
{
    /* Filter the message and see if action is needed */
    if ( gtimsg = GT_FilterIMsg( imsg ) )
    {
        switch ( gtimsg->Class )
        {
            /* Act depending on the message */
            ...
        }
        GT_PostFilterIMsg( gtimsg );
    }
}
/* other stuff can go here */
ReplyMsg( imsg );
```

It is essential that you make no assumptions about the contents of the unfiltered `IntuiMessage` (`imsg` in the above example). Only two things are guaranteed: the unfiltered `IntuiMessage` is guaranteed to be an `IntuiMessage` that needs to be replied to, and that when passed through `GT_FilterIMsg()`, the unfiltered message will produce a meaningful GadTools `IntuiMessage` like those described in the section on the different kinds of gadgets. The relationship between the unfiltered and filtered messages are expected to change. See the section on documented side-effects for more information.

10. *DrawBevelBox()*

A key visual signature shared by most GadTools gadgets is the raised or recessed bevelled box. Since you may wish to create your own boxes to match, GadTools provides the `DrawBevelBox()` function (and `DrawBevelBoxA()`, which takes a pointer to a `TagItem` array). `DrawBevelBox()` requires a pointer to the `RastPort` into which the bevelled box is to be rendered, as well as the dimensions (left, top, width, and height) of the desired box.

A bevelled box may either be raised (to signify an area of the window which is selectable), or recessed (to signify an area of the window in which clicking will have no effect).

`DrawBevelBox()` recognizes the following tags:

- ☐ `GT_VisualInfo` (APTR) - The `VisualInfo` handle as returned by a prior call to `GetVisualInfo()`. This value is required.
- ☐ `GTBB_Recessed` (BOOL) - Set to `TRUE` to get a recessed box, otherwise omit this tag entirely to get a raised box.

E. Restrictions on GadTools Gadgets

The gadgets created by GadTools should be handled with care. They are not designed to be bent, folded, stapled, or otherwise mutilated by wanton function calling. Put more succinctly, there is a strict set of functions and operations that are permitted on GadTools gadgets. Future releases of GadTools may broaden the set of allowed actions, but the restrictions listed here must be heeded. Even if you discover that something works for your particular case, be warned that it cannot be guaranteed to always be so. As well, you might be creating an illusion, if the trick you concoct only mostly works, but causes subtle problems down the line. If there is something you need to do to a GadTools gadget that isn't expressly permitted, contact Commodore-Amiga to see if we can supply an approved technique. If no such technique is possible, we may consider adding a way in a future release.

You must never selectively or forcibly refresh the gadgets. The only gadget refresh you should ever perform is the initial `RefreshGList()` and `GT_RefreshWindow()` when you first add your gadgets to the window. You should never need to use these kinds of functions at other times.

You may not selectively remove or add GadTools gadgets to a window. This has to do with the number of Intuition gadgets that each call to `CreateGadget()` produces, and also has to do with refresh constraints. If you try, you may be disappointed now or in the future.

Never use `OnGadget()` or `OffGadget()` or directly modify the `GADGDISABLED` Flag bit. The only approved way to disable or enable a gadget is to use `GT_SetGadgetAttrs()` and the `GA_DISABLED` tag. Those kinds of GadTools gadgets that do not support `GA_DISABLED` may not be disabled with this version of GadTools, period.

You should not be writing into any of the fields of the Gadget structure, or any of the structures that hang off it, with the exception noted earlier for `GENERIC_KIND` gadgets. You should avoid making assumptions about the contents of these fields unless you own them (`GadgetID` and `UserData`, for example), or if they are guaranteed meaningful (`Left`, `Top`, `Width`, `Height`, `Flags`). On occasion, you are specifically invited to read a field, for example the `StringInfo->Buffer` field.

Not writing into the gadget structure means several things. GadTools gadgets may not be made relative sized or relative positioned (`GRELWIDTH`, `GRELHEIGHT`, `GRELBOTTOM`, and `GRELRIGHT`). You may not alter the activation (for example changing `GADGIMMEDIATE` to `RELVERIFY`). You may not modify the imagery or the highlighting method. These restrictions are not imposed without reason; subtle or blatant problems may arise now or in future versions of GadTools.

F. Documented Side-Effects

There are certain aspects of the behavior of GadTools gadgets that should not be depended on, since we can already envision ways in which they might change. To help you remain compatible with future releases of the operating system, here are some of the known side-effects that are likely to change.

If you use `GT_FilterMsg()` and `GT_PostFilterMsg()`, never make assumptions about the message before or after filtering. By this we mean you should not interpret the unfiltered message, nor assume that it will or will not result in certain kinds of filtered message, or whether it will result in a consumed message (i.e. `GT_FilterMsg()` returns `NULL`). This is extremely likely to change.

If you are expecting `INTUITICKS IDCMP` messages, be warned that they are consumed when a scroller's arrows are repeating. That is, you will not hear `INTUITICKS` while the user is pressing a scroller arrows. This will not necessarily remain so.

When you call `CreateGadget()`, one or more actual gadgets may get created. These gadgets, along with the corresponding code in GadTools, define the behavior of the particular kind of GadTools gadget you have requested. We document the behavior only. The number or type of actual gadgets you really get is subject to change. You should always refer to the gadget pointer you receive from `CreateGadget()` when you call `GT_SetGadgetAttrs()`. You never should worry about the others, nor create code which depends on their number or form.

For text-display gadgets, the `GTTX_CopyText` tag does not cause the text to be copied when you later change the text with `GTTX_Text`.

The `PLACETEXT ng_Flags` are ignored by `GENERIC_KIND` gadgets. This may not always be so.

All GadTools gadgets set `GADTOOL_TYPE` in the gadget's `GadgetType` field. Do not use this flag to identify GadTools gadgets - we cannot guarantee that it will always be set.

The palette gadget subdivides its total area into the individual color squares. Do not assume that the subdivision algorithm won't change.

We are looking at alternate ways to indicate which color is currently selected in a palette gadget. The appearance (but not the basic shape) of the palette gadget might change as a result.

V. GadTools Menus

The greatest difficulty in creating menus is that a large number of finicky structures must be filled out and linked. This is bothersome because much of the required information is orderly and is easier to do algorithmically than to do manually.

As well, there are a lot of details to worry about if you want to do sensible layout of menus. This includes some mechanical items such as font-sensitivity, automatic columnization of too-tall menus, and accounting for space for checkmarks and Amiga-key equivalents. As well, there are esthetic considerations, such as how much spacing to provide, where sub-menus should be placed, and so on.

The GadTools menu functions support all the features that most applications will need. This includes:

- ☐ An easily constructed and legible description of the menus.
- ☐ Font-sensitive layout.
- ☐ Support for menus and sub-menus.
- ☐ Sub-menu indicators (a ">>" symbol attached to items with sub-menus).
- ☐ Separator bars for sectioning your menus.
- ☐ Command-key equivalents.
- ☐ Checkmarked and mutually exclusive checkmarked menu items.
- ☐ Graphical menu items.

First, let us have a look at how a typical menu strip might be specified:

```
struct NewMenu mynewmenu[] =
{
    { NM_TITLE, "Project",      0 , 0, 0, 0,},
    { NM_ITEM,  "Open...",      "O", 0, 0, 0,},
    { NM_ITEM,  "Save",          0 , 0, 0, 0,},
    { NM_ITEM,  NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM,  "Print",         0 , 0, 0, 0,},
    { NM_SUB,   "Draft",         0 , 0, 0, 0,},
    { NM_SUB,   "NLQ",           0 , 0, 0, 0,},
    { NM_ITEM,  NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM,  "Quit...",       "Q", 0, 0, 0,},

    { NM_TITLE, "Edit",          0 , 0, 0, 0,},
    { NM_ITEM,  "Cut",            "X", 0, 0, 0,},
    { NM_ITEM,  "Copy",           "C", 0, 0, 0,},
    { NM_ITEM,  "Paste",          "V", 0, 0, 0,},
    { NM_ITEM,  NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM,  "Undo",           "Z", 0, 0, 0,},

    { NM_END, 0,                0 , 0, 0, 0,},
};
```

This NewMenu specification would produce two menus. The first, called "Project", would have items called "Open", "Save", "Print", and "Quit". The "Print" item would have two sub-items, namely "Draft" and "NLQ". The "Edit" menu would have "Cut", "Copy", "Paste", and "Undo" as menu items, with a separator bar just above "Undo". As well, the menu strip would have the command key equivalents shown ("O", "Q", "X", etc.)

The nice thing about NewMenu arrays is that they can be read easily. The elements in the NewMenu array appear in the same order as they will appear on-screen. There is no need to specify the sub-menus first, the menu items (with their sub-menus hooked in) next, and the menu headers (with their menu items hooked in) last. The indentation used above also helps underline the relationship between menus, menu items, and sub-items.

A. The NewMenu Structure

GadTools menus are specified by filling out an array of NewMenu structures. The NewMenu structure is defined in *<libraries/gadtools.h>* as

```
struct NewMenu
{
    UBYTE  nm_Type;
    STRPTR nm_Label;
    STRPTR nm_CommKey;
    UWORD  nm_Flags;
    LONG   nm_MutualExclude;
    APTR   nm_UserData;
};
```

The first field, nm_Type, defines what this particular NewMenu describes. NM_TITLE is used to signify a menu heading. To specify an item, use NM_ITEM. Sub-items are declared using NM_SUB. The last entry in the array must have NM_END in this field.

Note that this provides an unambiguous and convenient representation. Each NM_TITLE signifies a new menu panel. Each NM_ITEM becomes a menu item in that menu panel. All the consecutive NM_SUBs that follow a menu item (NM_ITEM) compose that item's sub-menu panel. A subsequent NM_ITEM would be the next item in the original panel, while a subsequent NM_TITLE would begin the next menu panel.

NM_TITLE, NM_ITEM and NM_SUB are used for textual menu headers, menu items and sub-items respectively, in which case nm_Label points to the string to be used. This string is not copied, but rather a pointer to it is kept. Therefore the string must remain valid for the active life of the menu. GadTools also supports graphical menu items and sub-items (Intuition does not allow graphical menu headers). Simply use IM_ITEM and IM_SUB instead, and point nm_Label at a valid Image structure.

Sometimes it is a good idea to put a separator between sets of menu items or sub-items in the same panel. You may wish to separate the more drastic menu items ("Quit" or "Delete", for example) from the more mundane ones. GadTools will give you a separator bar if you supply the special constant NM_BARLABEL for the nm_Label field of an NM_ITEM or NM_SUB.

The NewMenu structure has the nm_CommKey field, where you may place a single-character string that is to be the Amiga-key equivalent you want for that menu item or sub-item. (Menu headers cannot have command keys.) Note that assigning a command-key equivalent to a menu item that has sub-items is meaningless and should be avoided. Notice that nm_CommKey is a pointer to a string, and not a character itself. This was done in part because routines to support different languages typically return strings, not characters. The first character of the string is actually copied into the resulting MenuItem structure.

The nm_Flags field of the NewMenu structure corresponds roughly to the Flags field of the Intuition Menu and MenuItem structures. However, for your convenience the sense of the Intuition MENUENABLED and ITEMENABLED flags are inverted. When using GadTools, menus and menu items (and sub-items) are enabled by default. To specify a disabled menu, set the NM_MENUDISABLED flag in this field. To disable an item or sub-item, set the NM_ITEMDISABLED flag.

As well, the Intuition flag bits COMMSEQ, ITEMTEXT, and HIGHFLAGS (which correspond to whether this item has a command-key equivalent, whether it is textual or graphical, and what method of highlighting is to be used) will be set for you, so do not set these in nm_Flags.

The nm_Flags field is also used to specify checkmarked menu items. To get a checkmark that the user can toggle, set the CHECKIT and MENUTOGGLE flags in the nm_Flags field. If the item or sub-item is to start its life in the checked state, also set the CHECKED flag.

Intuition also supports mutual exclusion of checkmarked items, and GadTools gives you easy access to that, too. All the items (or sub-items) that are part of a mutually exclusive set should have the CHECKIT flag set. Those that are to start out checkmarked should also have the CHECKED flag set. The nm_MutualExclude field of the NewMenu structure is a bit-wise representation of the items (in the same menu panel or sub-menu panel) excluded by this one. In the simple case of mutual exclusion (each choice excludes all others), set nm_MutualExclude to $\sim(1 \ll \text{item number})$, or ~ 1 , ~ 2 , ~ 4 , ~ 8 , etc. Do not forget that separator bars count as items too. See the Intuition Menus chapter in the ROM Kernel Manual for full details on menu mutual exclusion.

For your convenience, the NewMenu structure also has an nm_UserData field. This data is transferred to a special place immediately following the Intuition Menu or MenuItem structures that GadTools creates. Use the GTMENU_USERDATA(menu) and GTMENUITEM_USERDATA(menuitem) macros in *<libraries/gadtools.h>* to extract or change the UserData fields of Menus and MenuItems respectively.

There are several good uses for a menu's `UserData`. You could put index numbers there and perform a "switch" statement on them, instead of using the Intuition menu numbers. The advantage of this is that the numbers you choose remain valid if you rearrange your menus, while the Intuition menu numbers may change. You may then also use these numbers as indexes into a string-database, to achieve language independence in your menus. Alternately, an efficient technique is to have a specific handler function for each menu item, and put a pointer to that function in the corresponding item's `UserData` field. When you receive a `MENUPICK` message, you would call the selected item's function.

B. Functions for GadTools Menus

1. `CreateMenus()`

The `CreateMenus()` function takes an array of `NewMenu` and creates a set of initialized and linked Intuition `Menu`, `MenuItem`, and `IntuiText` structures, that need only to be formatted before being used. Like the other tag-based functions, there is a `CreateMenusA()` call that takes a pointer to an array of `TagItems`, and a `CreateMenus()` version that expects to find its tags on the stack. The first parameter that `CreateMenus()` takes is a pointer to the array of `NewMenu` structures described earlier.

Currently, `CreateMenus()` recognizes only one tag, namely `GTMN_FrontPen`, which is the pen number to use for menu text and separator bars. This tag has a default value of zero.

`CreateMenus()` returns a pointer to the first `Menu` structure that is created, while all the `MenuItem` structures and any other `Menu` structures hang off the appropriate pointers. If the creation fails for some reason (usually due to a lack of memory), `CreateMenus()` will return `NULL`. If your `NewMenu` structure begins with an entry of type `NM_ITEM` or `IM_ITEM`, then `CreateMenus()` will return a pointer to the first `MenuItem` created, since there will be no "first" `Menu` structure. Before you add the menus to your window (with Intuition's `SetMenuStrip()` function) you must lay them out with `LayoutMenus()`.

2. `LayoutMenus()`

The `Menu` and `MenuItem` structures returned by `CreateMenus()` contain no size or positional information. This information is added in a separate layout step, using `LayoutMenus()`. `LayoutMenus` takes a pointer to a `Menu` structure (the result of `CreateMenus()`), a `VisualInfo` handle (that you obtained from `GetVisualInfo()`), and a set of tags, and fills in all the size, font, and position information for your menu strip. As with the other tag-based functions, you may call either `LayoutMenus()` or `LayoutMenusA()`. `LayoutMenus()` returns `TRUE` if successful, and `FALSE` if it fails. The usual reason for failure is that the font you supplied cannot be opened.

`LayoutMenus()` recognizes a single tag, `GTMN_TextAttr`, which is a pointer to an openable font (`TextAttr` structure) to be used for the menu item and sub-item text. By default, the screen's font will be used.

`LayoutMenus()` takes care of calculating the width, height, and position of each menu item and sub-item, as well as positioning the menu panels and the sub-menu panels. (Note that you should not also call `LayoutMenuItems()` after having called `LayoutMenus()`). In the event that a menu panel would be too tall for the screen, it is broken up into multiple columns. As well, whole menu panels may be shifted left from their natural position to ensure that they fit on-screen. If you have a large menu combined with a large font, it can happen that even with columnization and shifting the panel is too big for the screen. Unfortunately, GadTools does not currently provide any protection against that.

It is perfectly acceptable to re-layout the menus by calling `ClearMenuStrip()` to remove the menus, `LayoutMenus()`, and then `SetMenuStrip()`. You would do this if you want to change the menu's font (which is a tag to `LayoutMenus()`) or if you want to change the menu's text (say to a different language). Later, we will describe how to do run-time language switching in menus.

3. *LayoutMenuItems()*

LayoutMenuItems() is a similar function to *LayoutMenus()*, but only affects the menu items (and their sub-items) of a single menu panel. This function is useful if you have an extensible menu (such as the Workbench's "Tools" menu). You may for example create a single MenuItem by calling *CreateMenus()* with a two-entry NewMenu array whose first entry is of type NM_ITEM and whose second is of type NM_END. You can then remove the menu strip, link this new item to the end of your extensible menu, call *LayoutMenuItems()* for that menu, and re-attach the menu strip.

LayoutMenuItems() takes a pointer to the first MenuItem to be formatted, the VisualInfo handle you have obtained from *GetVisualInfo()*, and a set of tags (either an array passed to *LayoutMenuItemsA()*, or stack-based tags passed to *LayoutMenuItems()*). In addition to recognizing the same GTMN_TextAttr tag that *LayoutMenus()* accepts, *LayoutMenuItems()* also knows the GTMN_Menu tag. Use this tag to provide the pointer to the Menu structure whose FirstItem you have passed as the first parameter to this function. We recommend that you always provide GTMN_Menu.

LayoutMenuItems() returns TRUE if it succeeds, and FALSE otherwise.

4. *FreeMenus()*

FreeMenus() will free all the memory allocated by the corresponding call to *CreateMenus()*. Its parameter is the Menu (or MenuItem) pointer that was returned by *CreateMenus()*. It is safe to call *FreeMenus()* with a NULL parameter.

C. Restrictions on GadTools Menus

GadTools menus are regular Intuition menus. Once the menus have been laid out, you may do anything you like with them, including attaching them or removing them from windows, enabling or disabling items, checking or unchecking checkmarked menu items, etc. See the Intuition documentation for *SetMenuStrip()*, *ClearMenuStrip()*, *ResetMenuStrip()*, *OnMenu()*, and *OffMenu()*, as well as the Intuition Menus chapter of the ROM Kernel Manual for full details.

If a GadTools-created menu strip is not currently attached to any windows, you may change the text in the menu headers (Menu->MenuName), the command-key equivalents (MenuItem->Command) or the text or imagery of menu items and sub-items, which can be reached as

```
( ( struct IntuiText * )MenuItem->ItemFill )->IText  
or  
( ( struct Image * )MenuItem->ItemFill )
```

You may also link in or unlink menus, menu items, or sub-items, if you wish. However, do not add sub-items to a menu item that was not created with sub-items, and do not remove all the sub-items from an item that was created with some. Also, Intuition does not like a menu header which has no items.

You may make any of these changes, provided you subsequently call *LayoutMenus()* or *LayoutMenuItems()* as appropriate. Then, re-attach your menu strip using *SetMenuStrip()*.

Some of these manipulations mean that you must walk the menu strip using the usual Intuition-specified linkages. Beginning with the first Menu structure, simply follow its FirstItem pointer to get to the first MenuItem. The MenuItem->SubItem pointer will lead you to the sub-menus. MenuItems are connected via the MenuItem->NextItem field. Successive menu panels are linked with the Menu->NextMenu pointer.

D. Language-Sensitive Menus

If you want your application to be able to switch the language displayed in your menus, it can be done quite easily. Simply detach the menu strip and replace the strings in the IntuiText structures as described above. It may be convenient to store some kind of index number in the Menu and MenuItem UserData which you can use to retrieve the appropriate string for the desired language. After all the strings have been installed, call LayoutMenus() and SetMenuStrip().

VI. Future Directions

The 2.0 release of AmigaDOS is the first release to incorporate the Gadget Toolkit. From here, we can see many directions in which GadTools should grow. Among the possible enhancements to gadgets are:

- ☐ Support for GadTools gadgets in Intuition requesters.
- ☐ Checkbox and radio button glyphs of arbitrary size.
- ☐ Make all GadTools gadgets obey the GA_DISABLED tag.
- ☐ More support to help in font- and language-sensitive gadget layout.
- ☐ A better method of indicating the selected color in palette gadgets.
- ☐ Replacing cycle gadgets by pop-up menus.
- ☐ A fancier listview, supporting such things as multiple selection, highlighting of the current selection, editing in the list, etc.
- ☐ Easing of some of the restrictions explained earlier. A number of restrictions may be removed just by converting GadTools to use Intuition's BOOPSI object-oriented custom gadget system.

Certain kinds of gadgets may grow additional features and functions. As well, some of the imagery and behavior may be improved or altered. Of course, any side effect may change in behavior.

The primary enhancements we can imagine for menus have to do with supporting cases when the menu panels end up too large. GadTools may be able to provide some error information back to the caller to inform the application that the menus are too big, and perhaps even by how much. The application would then attempt to lay out the menus with a smaller font.

It must be stated that none of these ideas constitute any kind of a promise. As well, we are not suggesting how soon we may get around to any or all of these. The list is given here to show you what we are thinking.

VII. Conclusion

The Gadget Toolkit provides a convenient, flexible, and powerful means of creating a sophisticated user-interface for your Amiga applications. A GadTools interface is easy to design and program, and it is easy for your users to grasp. With GadTools, we can all benefit from software that can be completed sooner, and looks and behaves better. ♦



Using New DOS Calls - Why and How

by Randell Jesup

1. Introduction

A common issue facing many developers is which new DOS functions to use, and how to use them. This paper should give some suggestions and hints towards making effective use of the new functionality which is available in 2.0. Not every function is needed by every developer, nor should a developer use a new function instead of his own version in every case. Hopefully, though, the new functions should make it easier and quicker for you to write programs for 2.0, or make it possible to write applications which weren't possible before.

2. Quick Overview

This is a quick listing of the general classes of new functions available to you under the 2.0 DOS:

- ☐ File Change Notification
- ☐ Cooperative Record Locking
- ☐ Standardized Command-Line Processing
- ☐ Standardized Pattern-Matching
- ☐ Simple Buffered Input/Output Routines
- ☐ More Flexible Handling of Locks and FileHandles
- ☐ Atomic Directory Scanning
- ☐ Easy "Thread Process" Creation
- ☐ Support for 3rd-party Shells, System Call
- ☐ Many Support Functions for Applications, Shells or Commands

Examples for most of these should be on the example disks.

3. Compatibility

As noted at the '89 Devcon, the DOS has been rewritten in C and some assembler, while maintaining compatibility with the old BCPL commands and applications for the time being. Certain types of rule-bending or breaking are no longer supported (this is why the pre-2.0 Ed doesn't work under 2.0, unfortunately). An example of this is calling BCPL global vector routines by using JSR _LVOOpen+2(A6). This will *not* work under 2.0, as DOS now uses a normal library jump table.

Applications which play by the rules (and even many that don't) should continue to work with the 2.0 DOS. A number of structures have been expanded; where you see "private" in the comments for a field, do *not* access it in *any* way if you wish to continue to work under future versions. Use the functions provided instead.

Most public or semi-public structures have remained the same. You are encouraged to try to make use of functions where possible, and to avoid certain semi-public fields, such as the fl_Link field of locks, since safety of access (and even the type of the contents) cannot be guaranteed (and never could be). This mostly affects debugging utilities such as ShowLocks. For debugging and similar utilities, doing things like this can be overlooked, so long as the authors understand (and communicate to their users!) that the operation of the program is unsafe or relies on things that may change.

For those trying to write programs that run under 1.3 as well as 2.0, you can use version checks to decide to make new calls, or duplicate some of the functionality of 2.0 functions. It will be a (hopefully little) while before most of you start requiring 2.0, but you can provide some added functionality in the meantime.

4. Notification

One of the most interesting and potentially most powerful additions is file change notification. This is the ability to have your application notified by message or signal when a file is changed in any manner.

Why and where should Notification be used?

One possible use of this is *hot links* between applications. The 2.0 Preferences programs use notification to communicate with interested parties (the system or an application). For example, when the font selection program writes a new default font file to ENV:sys, your program can be notified of this and can adopt the user's new preferred font. You can write your own configuration/preferences editors for your applications that do similar things without having to either have built-in configuration editing or external text files that are read at startup or upon user request.

To give another example of this ability, a page-layout program could be notified of changes to objects included in a page, and update automatically. The user might flip a draw program to the front, and revise an image which had been imported to the page-layout program. When the draw program writes out the image, the page-layout program would get a notification message, reread the file, and update the page for the user.

This is just a quick, simple example of some of the possibilities this opens up. I expect that you, the developers, can come up with some far more inventive ways to utilize this feature. Discuss this, think on it, and talk with other developers to come up with new uses and standards for those uses.

How to use Notification

To use notification, an application first does a StartNotify(). It will then receive messages when the file changes. Before exiting, the application should call EndNotify(). Messages (or signals) will be sent if the file is written to, created, deleted, truncated, or has its date modified ("touched"). Nothing is sent when the protection bits or file comments are changed (or at least nothing is guaranteed to be sent).

Due to an oversight, NotifyRequests aren't created by AllocDosObject under 2.0. This will be added to the next release. When you initialize your NotifyRequest, remember to set nr_Name to the name of the object, *not* nr_FullName (which is for DOS to use). Initialize nr_Flags to one of NRF_SEND_MESSAGE or NRF_SEND_SIGNAL, and initialize the union as appropriate. The references to COPY_DATA are historical, you should ignore those fields.

Most applications should also set the `NRF_WAIT_REPLY` flag if they're using notification by message. This means that the handler will not send your application another notification for that file until you reply the previous one. If the file had changed one or more times before you replied, it will notify you again immediately.

`NRF_NOTIFY_INITIAL` is useful to avoid special-case processing during startup. It causes the handler to send you a change notification when you do `StartNotify()` if the file exists already. This can save you from having to try to read the file explicitly in your startup; you can just let your normal notification handling in your main loop notice it.

After a successful `StartNotify()`, you should *not* modify your `NotifyRequest` until you do an `EndNotify()` (it's the property of the handler). When you wish to stop notification, call `EndNotify()`. `EndNotify()` will also scan the messages waiting at the `MsgPort` (for `NRF_SEND_MESSAGE`) and reply any notifications for this `NotifyRequest`.

A final word of caution: not all handlers can support notification. In particular, network handlers (such as NFS) are unlikely to support it. Check the return codes from `StartNotify()`, and if possible don't make the proper functioning of your program require notification. Using the above example, in your page-layout program, allow the user to tell you re-read an image (while doing it automatically if you can).

5. Cooperative Record Locking

DOS and the filesystems now include a simple form of record locking. This record locking is voluntary (cooperative), and is based on Novell formats for specifying record locks. The filesystems do not stop other processes from reading or writing a locked area, but they do stop other people from locking it when they shouldn't. You can lock one or more records, and timeouts are supplied. Locks can be shared (read) or exclusive (usually for write).

Why and where should Record Locking be used?

Record locking is useful (perhaps required) when you expect more than one process to be updating a portion of a file. Of course, it's particularly useful for database programs, especially ones that expect to run in a networked environment. Such environments will become far more common in the future.

Having a record locked does *not* stop other processes from reading or writing the data. However, it does stop other processes from locking it (depending on the modes involved). This is why it's *cooperative* record locking.

How to use Record Locking

Record Locking is quite easy to use. To lock a single record, use `LockRecord`. The timeout parameter is in unit of "ticks" (1/50 of a second). If the record is not available by the time specified, `LockRecord` will return with an error (see `dos/dos.h` for error codes). You can also specify that no timeout should be used (modes `REC_SHARED_IMMED` and `REC_EXCLUSIVE_IMMED`), and that if the record is not available it return immediately.

You can lock a record in exclusive or shared modes. Any number of people can lock a record in shared mode, while if someone has it locked in exclusive mode, no one else can have it locked. Normally, these are thought of as read and write modes. Exclusive locks that are waiting for others to unlock cause all requests for the area they cover to wait behind them.

You can request that a series of records be locked. This is done by filling out a `RecordLock` structure (see `dos/record.h`), and calling `LockRecords`. `LockRecords` takes a `RecordLock` pointer and a timeout. It will attempt to lock the records in sequence, using timeout on each lock attempt. If one of the attempts fails, all previously obtained locks are freed and a failure is reported. To avoid "deadly embraces" (deadlocks) you should try to lock the records in the same order. In record locks, deadly embraces aren't fatal, since one or the other (perhaps both) will time out eventually. Don't set infinite timeouts when this is possible!

Unlocking records is done with `UnLockRecord` and `UnLockRecords`. You should remove all record locks before closing the filehandle.

6. Standardized Command-Line Processing

The same routine used by all the Shell commands is available to applications. `ReadArgs()` prints the familiar prompt when you type "command ?". It allows specifying a fairly complex set of command-line options, and does some of the validation for you. It also can be used to parse strings from other sources.

Why and where should `ReadArgs` be used?

If you expect to require 2.0 for your application, and your program takes any arguments from the shell, then you should use `ReadArgs` to parse your arguments if possible. This gives a consistent and comfortable user interface to users, instead of having to become familiar with many different command-line syntaxes and quirks.

How to use `ReadArgs`

Since there are a number of different ways to use `ReadArgs`, I'll detail a common way of using it.

`ReadArgs` parses the command line according to a template that is passed to it. This specifies the different command-line options and their types. A template consists of a list of options. Options are named in "full" names where possible (for example, "Quick" instead of "Q"). Abbreviations can also be specified by using "abbrev=option" (for example, "Q=Quick").

Options in the template are separated by commas. To get the results of `ReadArgs`, examine the array of longwords you passed to it (1 entry per option in the template). This array should be cleared (or initialized to your default values) before passing to `ReadArgs`. Exactly what is put in a given entry by `ReadArgs` depends on the type of option. The default is a string (a sequence of non-whitespace characters, or delimited by quotes, which will be stripped by `ReadArgs`), in which case the entry will be a pointer.

Options can be followed by modifiers, which specify things such as the type of the option. Modifiers are specified by following the option with a '/' and a single character modifier. Multiple modifiers can be specified by using multiple '/'s. Valid modifiers are:

- /S - Switch.** This is considered a boolean variable, and will be set if the option name appears in the command-line. The entry is the boolean (0 for not set, non-zero for set).
- /K - Keyword.** This means that the option will not be filled unless the keyword appears. For example if the template is "Name/K", then unless "Name=<string>" or "Name <string>" appears in the command line, Name will not be filled.
- /N - Number.** This parameter is considered a decimal number, and will be converted by ReadArgs. If an invalid number is specified, an error will be returned. The entry will be a pointer to the longword number (this is how you know if a number was specified).
- /T - Toggle.** This is similar to a switch, but when specified, it causes the boolean value to "toggle". Similar to /S.
- /A - Required.** This keyword must be given a value during command-line processing, or an error is returned.
- /F - Rest of line.** If this is specified, the entire rest of the line is taken as the parameter for the option, even if other option keywords appear in it.
- /M - Multiple strings.** This means the argument will take any number of strings, returning them as an array of strings. Any arguments not considered to be part of another option will be added to this option. Only one /M should be specified in a template. Example: for a template "Dir/M,All/S" the command-line "foo bar all qwe" will set the boolean "all", and return an array consisting of "foo", "bar", and "qwe". The entry in the array will be a pointer to an array of string pointers, the last of which will be NULL.

There is an interaction between /M parameters and /A parameters. If there are unfilled /A parameters after parsing, it will grab strings from the end of a previous /M parameter list to fill the A's. This is used for things like Copy ("From/A/M,To/A").

ReadArgs returns a struct RArgs if it succeeds. This serves as an "anchor" to allow FreeArgs to free the associated memory. You can also pass in a struct RArgs to control the operation of ReadArgs (normally you pass NULL for the parameter, and ReadArgs allocates one for you). This allows providing different sources for the arguments, providing your own string buffer space for temporary storage, and extended help text. See *dos/rdargs.h* for more information on this. Note: if you pass in a struct RArgs, you must still call FreeArgs to release storage that gets attached to it, but you are responsible for freeing the RArgs yourself.

Here's a bit of sample code to illustrate using ReadArgs. A complete example should appear on your example disks.

```
/* This is an example of the command-line processing for a print program */
...

#define TEMPLATE          "Files/M/A,Header/K,Spool/S,Page=PageNumber/K/N"
#define OPT_FILES          0
#define OPT_HEADER         1
#define OPT_SPOOL           2
#define OPT_PAGE            3
#define OPT_COUNT          4

LONG opts[OPT_COUNT];      /* C guarantees this will be all 0's! */

...

struct RArgs *argsptr;
char **sptr;

...
/*=====*/
/* If ReadArgs() sees anything but zeros passed to it in elements */
/* of this array, ReadArgs() will assume that they are defaults. */
/*=====*/
argsptr = ReadArgs(TEMPLATE, opts, NULL);

/*=====*/
/* argsptr will be NULL if ReadArgs() failed, the secondary result */
/* code is fetched by IoErr(). */
/*=====*/
if (argsptr == NULL)
{
    PrintFault(IoErr(), NULL);      /* prints the appropriate err message */
}
else
{
    sptr = opts[OPT_FILES];
    if (!sptr)
        /* this can never actually happen, due to /A */
        VPrintf("No files specified!\n", NULL);
}
```

```

else
{
    VPrintf("files specified:\n",NULL);

    /* last string ptr is NULL */
    while (*sptr)
    {
        /* VPrintf takes a ptr to an array of arguments! */
        VPrintf("/t%s/n",sptr);
        sptr++;
    }
}

/* if option was not specified, it will be NULL (since buffer started */
/* with opt[] array all NULL).
   */

if (opts[OPT_HEADER])
{
    VPrintf("Header is '%s'\n",opts[OPT_HEADER]);
}

if (opts[OPT_SPOOL])
{
    VPrintf("Spooling selected\n");
}

if (opts[OPT_PAGE])
{
    /* the actual number can be accessed by
       *((long *) opts[OPT_PAGE])
       */
    VPrintf("Asked to print page %ld\n",opts[OPT_PAGE]);
}

...
/* cleanup */
FreeArgs(argsptr);
}
...

```

7. Standardized Pattern-Matching

Routines for both string pattern matching and directory wildcards are now available. These make accessing and manipulating pattern-matched files an easy operation. These routines are used by all the commands that allow patterns (most commands). They can also be used to search subdirectories as well as the current directory (as used in "dir all", etc).

Why and where should Pattern-Matching be used?

The pattern-matching routines (MatchFirst, MatchNext, MatchEnd) are useful wherever an application needs to access files named in a certain way, or where a user needs to specify patterns for objects to work on. The raw pattern routines (ParsePattern, MatchPattern) are useful inside applications when you want to allow the user to specify a pattern for some sort of selection from strings.

There are several advantages in using these routines over private pattern-matching routines. The size of executable files will be smaller. Development time will be cut as less code has to be written. Also, having a standard method of pattern-matching will assist in providing a consistent user interface.

How to use Pattern-Matching

The patterns are fairly extensive, and approximate some of the ability of Unix/grep regular expression patterns. Here are the available tokens:

?	Matches a single character.
#	Matches the following expression 0 or more times.
(ab/cd)	Matches any one of the items separated by ' '.
~	Negates the following expression.
[abc]	Character class: matches any of the characters in the class.
a-z	Character range (only within character classes).
%	Matches 0 characters always (useful in "(foobar%)").
*	Synonym for "#?", not available by default in 2.0. Available as an option that can be turned on.

"Expression" in the above table means either a single character (ex: "#?"), or an alternation (ex: "#(abcdlef)"), or a character class (ex: "#[a-zA-Z]").

To pattern-match a string, you must first call `ParsePattern`. This creates a tokenized version of your string (into a buffer you supply) that is used by `MatchPattern`. It also returns whether or not the string contained any wildcards. You then call `MatchPattern` to find if a given string matches the tokenized pattern.

To find files that have names matching a pattern, you call `MatchFirst`, `MatchNext`, and `MatchEnd`. These are passed an `AnchorPath` structure (see *dos/dosasl.h*) as a control structure.

`MatchFirst` is passed your pattern (you do not pass it through `ParsePattern` - `MatchFirst` does that for you) and the control structure. `MatchFirst` normally initializes your `AnchorPath` structure for you, and returns the first file that matched your pattern, or an error. Note that `MatchFirst/MatchNext` are unusual for DOS in that they return 0 for success, or the error code (see *dos/dos.h*), instead of having the application get the error code from `IoErr()`.

`MatchNext` gets the next file or directory that matches the pattern, and returns the same codes as `MatchFirst`.

When looking at the result of `MatchFirst/MatchNext`, the `ap_Info` field of your `AnchorPath` has the results of an `Examine` of the object. You normally get the name of the object from `fib_FileName`, and the directory it's in from `ap_Current->an_Lock`. To access this object, normally you would temporarily `CurrentDir` to the lock, do an action to the file/dir, and then `CurrentDir` back to your original directory.

There are various bits that control whether `MatchNext` should "enter" subdirectories and other functions, see *dos/dosasl.h*.

After getting `ERROR_NO_MORE_ENTRIES` (or any other error, for that matter), you must clean up by calling `MatchEnd`, which frees storage and any locks associated with the `AnchorPath`, after which you can free your `AnchorPath`.

8. Simple Buffered Input/Output Routines

A series of functions to do simplistic buffered I/O is available in 2.0 DOS. These are similar to ANSI C stdio routines, though not identical. They implement read/write buffers to reduce the amount of overhead in doing single character I/O.

Why and where should Buffered I/O be used?

The buffered I/O routines are useful in producing small programs that do not link in large amounts of HLL library code, or for people writing in assembler that don't want to write their own buffered I/O routines. They are far more efficient than doing single-character or small string reads or writes.

You can use them in any place you would use C stdio functions. This will mean your program will require 2.0, of course. Single and multi-character reads and writes, line reads and writes, equivalents of fprintf, etc, and record read/write routines are available. In addition, old BCPL-style formatting is available, though not recommended. By using the Flush function, you can mix buffered I/O with unbuffered I/O.

How to use Buffered I/O

Using buffered I/O is quite simple. Use the calls as you would the stdio equivalents (note that ordering of parameters is sometimes different). The single buffer will switch between read and write as needed, maintaining the current effective position in the file. Flush will cause unwritten write data to be written, unread read data in the buffer to be dropped, and the file position adjusted back to the right spot via Seek.

When switching between buffered and unbuffered I/O, you should call Flush to make sure the position and integrity of the file is maintained. You should call it when switching in either direction.

An important point: if you are using buffered I/O on your original default Input() filehandle, you *must* call Flush before reading any data. This is because shells have to stuff the buffer of the filehandle with the contents of the command-line to be compatible with 1.3 BCPL programs. Taking advantage of this fact is *not* encouraged. Use GetArguments instead.

Any unwritten write data in the buffer will be written when the file is closed, and the buffer deallocated.

SetVBuf currently doesn't do anything very useful, but we plan to allow different buffering modes in future releases, as well as the ability to change the size of the buffer. Currently, the buffer is flushed on writes of \n, \0, \r, or \12 for interactive filehandles, or of course when the buffer is full.

9. More Flexible Handling of Locks and FileHandles

There are a number of new routines that act on locks and filehandles. These greatly increase the flexibility available in manipulating objects. In particular, a number of previously impossible operations are now supported, such as examining an open filehandle. Some of them were possible, but were hard to code. Here's a quick list of some of them:

```
NameFromLock()  
NameFromFH()  
ExamineFH()  
DupLockFromFH()  
ParentOfFH()  
OpenFromLock()  
SameLock()  
SetMode()  
ChangeMode()  
SetFileSize()  
Inhibit()  
Format()  
Relabel()  
AddBuffers()
```

Why and where should the new Lock and Filehandle routines be used?

These routines are useful in many places. SetFileSize allows either truncating or extending files. NameFromLock/NameFromFH are useful in many, many places. OpenFromLock allows you to exchange a lock for a filehandle on the same object. SetMode does an ACTION_SCREEN_MODE, which has existed since 1.2, but had to be sent as a packet before (same for Inhibit, Relabel, and AddBuffers). ChangeMode allows you to change the mode of a lock or filehandle (exclusive to shared, for example).

Some of these can be duplicated with some work, such as NameFromLock. If you must run under 1.3, this may be an option. Many of the others can not be safely duplicated.

How to use the new Lock and Filehandle routines

In general, using these routines is fairly simple, and the autodocs should explain most of these calls fairly well. Since most of these are based on new filesystem packet types, you should be careful to check for errors, such as ACTION_NOT_KNOWN, to deal with older filesystems which have not been upgraded yet (mostly older network filesystem implementations).

SameLock checks for ACTION_NOT_KNOWN, and falls back on an alternative method (checking the handler address and the fl_Key field).

10. Atomic Directory Scanning

There has always been a problem with the semantics of ExNext, in that you can get "funny" results if the directory is modified during the period you're reading it. Also, ExNext is not extremely fast, even with the new FS (or FFS), since at least 2, and often 4 or 6 or more task switches must occur per file.

This has been dealt with in 2.0 with the ExAll call. This allows you to get the entire contents of a directory in a single call.

Why and where to use ExAll?

ExAll can be used in any place you'd use ExNext. In addition, ExAll supplies you with some extra abilities, such as being able to control how much information is returned (to save ram/buffer space), and being able to accept or reject files based on either system pattern-matching routines or routines of your own.

ExAll is up to 3 or 4 times faster than ExNext for large directories, and avoids potential dangers in ExNext. It does require 2.0 to use, of course, so, if you must run under 1.3, you must either use a version check, stick with ExNext, or write your own version for 1.3 that uses ExNext. This is quite possible, and, in fact, the 2.0 DOS includes such code to handle older filesystems that do not support ExAll yet (such as network handlers).

How to use ExAll

To use ExAll, you need a control structure and a buffer for the data. The control structure *must* be allocated by AllocDosObject. Do not violate this if you wish to run under future versions of the OS. The data buffer is simply an area of memory. Before you call ExAll, you must make sure the eac_LastKey field is NULL.

You also must decide how much information you want returned. Each higher level of information includes all those below it. The minimum is just the file/dir names, the maximum includes essentially everything in a FileInfoBlock. See *dos/exall.h* for more information.

When ExAll returns, the return code is FALSE if there is no need to call it again, otherwise it returns TRUE. This handles the case of not enough buffer space to return all the entries. It's best to avoid calling ExAll too many times for a directory, since the directory might change in between calls, so a reasonably large buffer is a good idea. If ExAll returns non-zero, you *must* continue to call ExAll until it returns FALSE.

You must also check the value of `IoErr()` if `ExAll` returns `NULL`. If it's something other than `ERROR_NO_MORE_ENTRIES`, an abnormal error has occurred, and you should go to some error-handling code. Another thing you have to deal with is the number of entries returned. This is stored in `eac_Entries` in the control structure. You must handle 0 entries found gracefully.

The data in the buffer consists of subsets of `ExAllData` structure, depending on how much data you asked to be returned. Each entry is chained to the next by `ead_Next`, and the last entry has `ead_Next = NULL`.

Here's a partial example of how to use `ExAll`:

```
eac = AllocDosObject(DOS_EXALLCONTROL, NULL);
if (!eac)
    /* return error or some such */

...

/* Must be zero before calling ExAll the first time */
eac->eac_LastKey = 0;

do
{
    more = ExAll(lock, EADData, sizeof(EADData), ED_COMMENT, eac);
    if ((!more) && (IoErr() != ERROR_NO_MORE_ENTRIES))
    {
        /* ExAll failed abnormally - set some error code, return failure */
        break;
    }
    if (eac->eac_Entries == 0)
    {
        /* ExAll failed normally with no entries */
        /* ("more" is *usually* zero) */
        continue;
    }
    ead = (struct ExAllData *) EADData;
    do
    {
        /* use ead here */
        ...
        /* get next ead */
        ead = ead->ed_Next;
    } while (ead);

} while (more);

...

FreeDosObject(DOS_EXALLCONTROL, eac);

...
```

11. Easy "Thread Process" Creation

Process creation has been made easier, especially for those wishing to create *thread* processes from subroutines. You can now specify essentially all parameters of the environment for the subprocess. This is done by `CreateNewProc`, which takes a taglist to specify options.

Why and where should `CreateNewProc` be used?

`CreateNewProc` has a number of uses. It can be used as a replacement for `CreateProc` to give you added control of a few features. It can be used to create processes that are setup like Shell processes (CLI structure, etc.) for running subprograms that want a shell-like environment. And it can be used to start arbitrary code (usually subroutines) as a process.

Threads are a very powerful paradigm for programming on a shared- address machine like the Amiga. They allow you to easily separate your program into different tasks, for example to avoid the program becoming unresponsive when doing disk I/O, or to continue refreshing windows during lengthy operations, etc. Some applications spawn subprocesses now, but most are written single-threaded. Also, separating the application like this can often make the event loops of the threads much simpler (for example, they might be able to use synchronous I/O instead of asynchronous, etc).

How to use `CreateNewProc`

`CreateNewProc` takes a single parameter, a pointer to a taglist. You must specify one of `NP_Seglist` or `NP_Entry`. `NP_Seglist` takes a seglist (as returned by `LoadSeg`). `NP_Entry` takes a code ptr for the routine to call.

There are many options, as you can see by examining *dos/dostags.h*. The defaults are for a non-CLI process, with copies of your `CurrentDir`, `HomeDir` (used for `PROGDIR`), priority, consoletask, windowptr, and variables. The input and output filehandles default to opens of `NIL`, stack to 4000, and others as shown in *dostags.h*. This is a fairly reasonable default setting for creating threads, though you may wish to modify it (for example, to give a descriptive name to the process.)

If you spawn a thread from your process, you should make sure it exits before your main process does (since the code would get freed). Currently, you'll have to do it yourself, since `NP_NotifyOnDeath` and `NP_Synchronous` are not currently "hooked up". `NP_ExitCode` and `NP_ExitData` are hooked up, though, and can be used to send a message to your main process. Actually, for threads of code from your application, it's easy to have the thread send an "I'm dead" message to the main process. `ExitCode` is more useful when the code being run is not part of your application.

12. Support for 3rd-party Shells, System Call

There is now support for 3rd-party shells to be the default "User Shell" in a system-friendly manner. This shell will be started whenever the user uses "NewShell" or the Shell icon from WB. In addition, the new System() call allows the application to specify which shell the command should be sent to. (System() is a more-functional replacement for Execute().)

Why and where should System() be used?

System() is a replacement for Execute(). It allows using an input filehandle without having commands executed from it (a strangeness of how Execute() works). Only the command(s) specified in the command string are run. System() also can work in an asynchronous fashion (it doesn't wait for the command to execute before returning).

Normally, System() passes commands to the Boot shell (the built-in shell). A tag can specify that the command go to either the preferred User Shell, or a specific named Shell (from the SYSTEM resident list).

System() should be used where you need to specify a input filehandle, or wish the command(s) to be executed by another shell. In general, commands written by the user should be passed to the User Shell, and commands built by your application should be passed to the Boot Shell. This avoids problems with user shells, such as the fact that they may interpret the command line in a different way (different special characters, built-in commands, etc).

How to use System()

It's quite easy to use System(). You merely specify a command line, and some optional tags. Input and output for the commands defaults to the same as your application, unless you use tags to redirect it. The Shell defaults to the Boot Shell unless you use the SYS_UserShell or SYS_CustomShell tags.

SYS_Asynch makes the call return immediately. If you use this, you *must* specify both input and output filehandles, which will be closed by the shell when the command is done.

System() (when not asynchronous) returns the return code of the command, and sets IoErr() to the secondary result of the command.

13. Many Support Functions for Applications, Shells or Commands

There are a large number of support calls in DOS to make application programming easier. Some are mostly to support filesystems or 3rd party Shells. Many (or most) could be written for 1.3 if you need to.

A quick list of some of these functions:

- ☐ A full set of low-level Packet routines.
- ☐ Functions for dealing with Process and CLI structure fields.
(please use these if writing 2.0-specific code!)
- ☐ SplitName, for parsing paths.
- ☐ SetIoErr, to set what IoErr returns
- ☐ Various routines for printing error strings for different errors.
- ☐ The routines that put up most of the "please insert", etc messages.
- ☐ RunCommand, which shells use to start programs.
- ☐ Routines for creating and removing Assigns.
- ☐ Routines for dealing with the DOS device/volume/assign lists.
(You *must* use these if writing 2.0-specific code!)
- ☐ Date comparison and conversion routines.
- ☐ Low-level entries to LoadSeg, etc.
- ☐ Resident list handling routines.
- ☐ Support routines for dealing with command lines (ReadItem and FindArg).
- ☐ A string-to-long conversion routine.
- ☐ Routines for splitting or appending to paths.
- ☐ Local/Global variable/alias/etc routines.

The most interesting of these to applications are the variable routines, path routines, Process and CLI functions, and error string routines. See the autodocs for more information on these routines.

14. Summary

The 2.0 DOS provides a wealth of new features to help you improve or enhance your applications. Few people have a need for every new feature, but most of you will find some things that are immediately applicable. In some cases, this gives you opportunities that exist on no other system, such as notification, and the things that come from it.

A large part of the new DOS is oriented towards making the command and application writer's task easier. This may not affect you for a little while, but 2.0 will become the standard in the near future. When this happens, we hope the DOS will save you time in design, coding, and especially debugging, while helping all of us provide a more powerful and more consistent user interface. We have already seen this in the writing of the new commands for 2.0. ♦



Basic Object Oriented Programming System for Intuition

by Jim Mackraz

Introduction

- Level 1
- Level 2
- Level 3
- Goals

Chapter 1: Using boopsi

- Black Box Objects
- Transparent Base Classes
- Attribute Lists
- Attribute Access Modes
- Varargs Interface
 - (Example)
- Functions
- Interconnections
- Interconnections and Simple Models
- Advanced Models
- Idcmp and Other Notification Methods
- Gadget Groups
- Class Documentation Standards
- boopsi Images

Chapter 2: Custom Gadget Implementation

- Custom Gadget Hooks
- Custom Gadget Methods
 - GM_HITTEST
 - GM_RENDER
 - GM_GOACTIVE
 - GM_HANDLEINPUT
 - GM_GOINACTIVE

Chapter 3: OOP Overview and Terminology

- Objects, Methods, Messages, Classes
- Inheritance and Transparent Base Classes
- Private vs. Public Classes

Chapter 4: Writing boopsi Classes

- Root Class Methods
 - OM_NEW
 - OM_DISPOSE
 - OM_ADDTAIL
 - OM_REMOVE
 - OM_ADDMEMBER
 - OM_SET
 - OM_GET
 - OM_UPDATE
 - OM_NOTIFY

Chapter 5: Advanced Tagitem List Use

Chapter 6: Boopsi Future

Introduction

This document covers new support for creation, interconnection, and grouping of Intuition gadgets, and the implementation of "custom gadgets." The system has an object-oriented structure and extends usefully to managing other "custom" entities, such as images.

The new system, informally called boopsi, is designed to be programmed at three different levels.

Level 1

The simplest thing you can do with boopsi is to be a "boopsi user." This means that pre-existing gadget and image definitions are used to create gadgets for windows or requesters, optionally interconnected and grouped.

The "classes" of gadget, image, and other objects that might be used include the classes defined by Intuition, other public classes provided in packages (libraries) on disk, and private classes or custom gadget definitions linked into the program "blindly".

Using gadgets created from boopsi classes is convenient and powerful. You may do that without knowing anything about the machinery used to implement gadgets, shared classes, and the underlying object-oriented principles.

Chapter 1 below presents a "naive gadget user" programmer's guide to programming at this first level. Subsequent chapters describe the more sophisticated levels.

Level 2

At the next level, the programmer writes the code to create a custom gadget. A custom gadget is one which renders itself, handles input, and is otherwise managed by code which is not part of Intuition proper.

When you present a custom gadget to Intuition, you provide a pointer to a "function callback" Hook structure that Intuition will call for processing during various gadget operations.

Level 3

The most sophisticated level of programming boopsi is in creating a "gadget class" to provide custom gadget processing in a sharable and more powerful framework. Beyond providing the basic routines needed to define a custom gadget, a class implementation also supports gadget creation and deletion, interconnection between gadgets, and a unified mechanism of defining gadget properties, or "attributes." Classes may also be defined to create "non-gadgets", such as images and abstract interconnection, broadcast, or calculation objects.

Classes may be private or public. In the first case, the code implementing the class is linked in with your program, the class is not named (at least is not installed in Intuition's official shared class list), and constants and function code values are assigned from a "name space" that does not conflict with the values that might be assigned to public, shared classes.

Goals

This system was designed and implemented to provide more convenient and extensible programming of Intuition user interfaces. The system is not intended as the be-all, end-all object-oriented application development system.

The requirements of this system differ from those normally met by "real" object-oriented programming systems. For one, it needs to be small, both in code space and per-object data. It also has to be reasonably fast on modest hardware. Another difference is that the system is language-independent. If a language can implement an interrupt handler on the Amiga, it can be used to create a public class (although the code does *not* run as an interrupt, but it often must run as the input.device task). Unlike many other OOP systems, the inheritance mechanism is very loosely coupled, so that third parties will not need to recompile the implementations of classes when the classes they are based on are changed in implementation, instance data, or by additions of new identifier constant definitions. Another difficult problem is keeping names and ID's unique for shared classes arriving from different sources. To keep this problem tractable, there is a "public/private split" in the name space (at least the numeric values of constants). This allow you to use classes you implement or collect from others as private classes, without registering them to be granted a set of values that won't ever collide with other public classes. Lastly, a primary requirement of this system is that it be integrated with, and compatible with, existing Intuition operation.

For more explanation of the object-oriented principles in the system, see Chapter 3, *Overview and Terminology*.

Chapter 1: Using Boopsi

Black Box Objects

You create a gadget from a class by calling the function `NewObject()`. You pass it an ID string or pointer-handle for the class of the gadget (or other objects) you want created, such as `propgclass` to create a proportional gadget, or `"boxiclass"` to create a box image. This "dynamic allocation" of gadgets, images, and other data objects is in contrast to most examples of Intuition programming which relies on initialized global "static" data in the program load module.

It also reflects a "black box" approach to programming, which is also a new thing for Intuition gadgets. When you are returned a gadget or other data structure from `NewObject()`, use it for nothing except as a handle to the various manipulation functions. These functions include new ones to perform new boopsi operations such as `SetAttrs()` and `DisposeObject()`, but also old functions like `AddGList()`, and `RefreshGList()`, to remain compatible.

Transparent Base Classes

When you create a gadget by calling `NewObject()`, the returned pointer points to a data structure that looks just like the familiar Intuition Gadget structure. You should not rely on the expected behavior of any of the fields of this structure unless explicitly documented. It is best to avoid making assumptions about the meaning of the Gadget structure fields. When you create an Image object, you also are returned a pointer to something that starts off looking like a familiar structure, the Image structure, but again, you should use the programmatic interface to change or examine the values of the Image structure so that you do not shortcut the *interpretations* that the particular class may put on the values it stores in various fields. Strive to be blind.

Attribute Lists

Attributes are used to help specify, notify changes in, and examine the values a gadget or other object maintains. Each public attribute of an object in a class is identified by a 32-bit ID value called an *attribute ID*.

An attribute ID coupled with a 32 bit data value in a `TagItem` structure (from *utility/tagitem.h*) is called an *attribute-value pair*. An array of `TagItem` attribute-value pairs (or chain of arrays) is called an *attribute list* or *tag list*. Attribute pairs and lists are the vehicles for communicating attribute values and their changes.

You may specify an attribute list at the time you create an object, passing it to `NewObject()`, or you may subsequently set attributes using `SetAttrs()` (or `SetGadgetAttrs()`, about which more later). You may inquire the value of only one attribute at a time, using the function `GetAttr()`.

Attribute Access Modes

Some attributes may only be set when an object is created, such as the size of the buffer of a boopsi string gadget, or the vertical or horizontal freedom of a proportional gadget. Other attributes might not respond to `GetAttrs()`. Some attributes may be updated on the fly by interconnections with gadgets or other objects. Those are attributes which support *update access*.

Gadgets typically have some attributes that change when the gadget gets input from the user, such as the state of a button, the value of a slider, or the selected item in a list. The gadget may possess the capability of notifying interested parties via interconnections when these values change. Such an attribute has *notify access*.

The access mode of each attribute will be documented, as described later.

Varargs (Variable Number of Arguments) Interface

There are actually two ways to pass an attribute list to the functions that accept them as arguments. One of them is to pass the address of a pre-constructed array of `TagItems`. Often, it is more convenient or readable to create the array on the stack.

It works out especially nicely with "standard" C parameter passing conventions.

The actual entry point for `NewObject()`, for example, is

```
NewObjectA( class, classid, taglist );
```

(you only pass one of 'class' or 'classid')

where `taglist` is a (the address) an *array* of `TagItems` containing the desired attribute values that are to be used in creating the object.

A "varargs" version of the function is named `NewObject()`, and must be provided for each implementation of a language that can support it. We will provide this either in *amiga.lib* or as assembler source, or both.

As an example, here are two ways to create a simple proportional gadget.

Using the initialized array method:

```
struct TagItem  proptags[] = {
    {GA_Left,      80},
    {GA_Top,       14},
    {GA_Height,    58},
    {GA_Width,     20},
    {GA_RelVerify, TRUE},
    {PGA_Freedom,  FREEVERT},
    {PGA_Top,      0},
    {PGA_Visible,  25},
    {PGA_Total,    100},
    { TAG_END, }
};

struct Gadget *propgadget = NewObjectA( NULL, "propgclass", proptags );
```

Using the varargs method:

```
struct Gadget *propgadget = NewObject( NULL, "propgclass",
                                       GA_Left,      80,
                                       GA_Top,       14,
                                       GA_Height,    58,
                                       GA_Width,     20,
                                       GA_RelVerify,  TRUE,
                                       PGA_Freedom,   FREEVERT,
                                       PGA_Top,      0,
                                       PGA_Visible,   25,
                                       PGA_Total,    100,
                                       TAG_END );
```

Note how convenient it would be to use a more complicated expression to specify any of the position or dimension parameters, for instance.

A quick-and-dirty C implementation of NewObject() follows for language implementations with "standard" parameter conventions:

```
Object  *NewObject( Class *class, ClassID classid, ULONG tag1, ...)
{
    Object *NewObjectA(
        Class *class, ClassID classid, struct TagItem *ti);
    return ( NewObjectA( class, classid, (struct TagItem *) &tag1 );
}
```

Functions

Several functions are designed for use by simple class users. The autodocs for these functions go into more detail and are the official reference. We will list both varargs and array-based names for functions which take both forms.

`NewObject(class, classid, tag1, value1, ..., TAG_END)`

`NewObjectA(class, classid, attrlist)`

This function, briefly described above, creates an object from the class specified. The class can be specified two ways. If the 'class' parameter is non-NULL, then it must point to the Class data structure, presumably of a private class.

If the 'class' parameter is NULL, then the 'classid' field must be the ID (a null-terminated text) string of the public, shared class.

The attribute-value pairs are a taglist (as defined in *utility/tagitem.h*) specifying initial values of selected attributes for object of the given class. Attributes with either "New" or "Set" access modes will be recognized.

Note that all of the following functions apply *only* to objects which are created using this function.

`DisposeObject(object)`

This function frees any data object created by `NewObject()`, and associated data. Sometimes you provide pointers to data structures as values of some attributes. If these data are "granted" to the object, they will be freed when this function is called. The class documentation must identify such behavior.

`GetAttr(attrid, object, storage_ptr)`

This function is used to inquire an object's value for a given attribute. It returns non-zero if the attribute ID is recognized by the object's class (recognized attributes will be documented). The 'storage_ptr' is a pointer to memory storage of the specific type required (and documented) by the attribute.

`SetAttrs(object, tag1, data1, tag2, data2, ..., TAG_END)`

`SetAttrsA(object, attrlist)`

This function is used to specify a new set of attribute values for an existing object. Only those attributes documented as having "Set" access will be affected.

For some classes, changing an attribute value using this function will cause them to notify the objects they are connected to of the change. Gadgets, typically, only notify other objects when the user directly manipulates them resulting in a change of a *notify access* attribute value.

NOTE WELL: For technical reasons, a separate function is used when changing the attributes of a gadget or things connected to a gadget. This function is `SetGadgetAttrs()`, below.

`SetGadgetAttrs(object, window, requester, tag1, data1, ..., TAG_END)`

`SetGadgetAttrsA(object, window, requester, attrlist)`

This function is the same as `SetAttrs`, but also provides additional information that a gadget needs in order to draw itself in response to changed values (such as a change to the position of a proportional gadget slider).

It can be used with non-gadget objects just fine (they will ignore the gadget-specific information passed), and can be passed NULL as window and requester if the gadget is not yet attached to a window or requester.

This function also serializes changes to a gadget with the input processing and other operations that Intuition might apply to a gadget, so it is vital that this function be used in preference over SetAttrs() whenever a gadget object is involved.

`DoMethod(object, MethodID, param1, param2, param3)`

In object-oriented parlance, the functions that a class defines for the objects in that class are called "methods". These are implemented for a boopsi class as a single entry point which is passed a function code, or MethodID, along with various other parameters appropriate to the specific method.

We've seen some methods wrapped up in specific function calls, such as `DisposeObject()` (MethodID: OM_DISPOSE) and `SetAttrs()` (MethodID: OM_SET). This function serves as an interface to invoke other method functions that a class might define, such as adding an object to a list, or performing some other special operation. These additional methods must be documented clearly in the class definition to say if they are appropriate for application use (they may be only appropriate for internal or subclass implementation uses) and what restrictions or warnings must apply.

Invoking a method function defined by a given class, to be applied with an appropriate set of parameters to a given object of that class, is frequently called "sending a message to an object" in object-oriented terminology. This is a very intuitive metaphor, providing a concrete concept of an object as a black box, privately interpreting and acting on the various messages that come its way.

Unfortunately (or perhaps fortunately), the term "Message" is an "overloaded term" on the Amiga, conflicting with the Exec Message structure and the ARexx `RexxMessage`. During the implementation, we used the term `SendMessage` in disregard of these conflicts, but renamed this external interface function to `DoMethod()` later in development.

The function `DoMethod()` is a varargs version of a function which sends a prepackaged parameter structure (always starting with a MethodID) to the routines that implement the class from which the object was created. The pre-packetized version of `DoMethod()` is called `DM()`. Simple boopsi users will probably always use `DoMethod()`.

`DoMethod()` is implemented for standard C conventions in a file named `classface.asm` in the examples, along with some variations useful for class implementors.

Interconnections

Each gadget class may define some attributes with "Notify" access mode, which means that they will issue forth notification to interested parties when the value of that attribute changes due to user input.

The mechanism that supports this consists of two attributes recognized by "gadgetclass", ICA_TARGET and ICA_MAP. (ICA_ stands for "interconnection attribute", and we shall see that this aspect of gadgets is borrowed from the simpler interconnection class "iclass".)

The value of ICA_TARGET is some other object. The value of ICA_MAP is a TagItem list of a special kind, defining a mapping from attribute ID's of the source object to attribute ID's of the target.

The class "propgclass" (proportional gadgets) has an attribute named PGA_TOP, conveniently interpreted as the top line of scrolling text controlled by this prop gadget that should be displayed. (Other attributes specify the total number of lines and the number of lines visible at one time.)

As the user drags the slider knob, the internal value of PGA_TOP will change. Whenever it does change, the prop gadget will "notify it's target through the map list." Let's explore what that means.

Suppose the PGA_TOP value of a slider has changed to 5. Then the propgadget will issue a notification of the attribute-value pair { PGA_TOP, 5 }. This will be converted as an internal SetAttrs() function applied to the object given as PGA_TARGET.

But although the target object may well understand the meaning of "5", it probably will not recognize the tag PGA_TOP. Thus, the ICA_MAP taglist is used to convert the attribute ID to some other value, recognized by the target.

For example, if you specify the ICA_TARGET for a propgadget to be a boopsi string gadget (from "strgclass"), and an ICA_MAP list containing:

```
{ PGA_TOP, STRINGA_LongVal }
```

then when the propgadget issues notification of:

```
{ PGA_TOP, 5 }
```

the string gadget will receive an internal SetAttrs() call specifying

```
{ STRINGA_LongVal, 5 }
```

and the string gadget will automatically display the value 5, with no application involvement whatsoever.

Interconnection Class and Simple Models

The mechanism described provides a data-driven direct interconnection scheme that works great whenever the value 5 here makes the proper sense to both parties in the exchange of information. When this happy coincidence doesn't hold up and some conversion is necessary, the inevitable glue processing must happen somewhere.

Also, the connection a gadget supports is limited (in the interest of small gadgets) to a single target/map pair.

a new general class of objects has been introduced. It is called *Interconnections* or *ICs* (from the class *icclass* and its subclasses) and special cases of ICs called *Models* (from the class *modelclass* and its subclasses, itself a subclass of *icclass*). ICs are simple information forwarders. Their only attributes are ICA_TARGET and ICA_MAP, which work just as defined for gadgets, except that ICs don't issue any notifications on their own.

What a vanilla IC does is take any SetAttrs() it receives and forward it directly to its ICA_TARGET after converting the attrlist using its ICA_MAP. The usefulness of such a simple class of objects is two-fold. First, ICs are used as nodes in a *broadcast list* maintained by a model (see below). Secondly, specializations (subclasses) of *icclass* can perform intermediate calculations on the attributes coming in before they are sent out the other side to its ICA_TARGET. For example, a specialized IC might take the attribute value for a tag MY_INT1, multiply it by 10 and issue a notification of MY_INT10 with the result.

Thus, if we set the target of the propgadget in our earlier example to be this special IC, and the map to include { PGA_TOP, MYINT1 }, and the target of the special IC to be our string gadget, with map including { MYINT10, STRINGA_LongVal }, then whenever you drag the slider, a value 10 times of the slider's current PGA_TOP value will be displayed in the string gadget. This shows how ICs can be used to encapsulate glue code in a reusable form for a data-driven interconnection scheme.

A model (object from *modelclass* or its subclasses) is only a simple extension of the basic functionality of an IC but is the center of a whole philosophy of object-oriented user-interface programming.

In its simplest form, a model is just an IC (with ICA_TARGET and ICA_MAP), plus a "broadcast list" of additional objects which it will also notify. Each node on this list will receive an update message (an internal SetAttrs() call) of unmapped attributes from the model object. Typically, the nodes on this list are primitive ICs, which contain nothing more than an ICA_TARGET (such as a gadget) and the appropriate ICA_MAP.

An example of the broadcast property of the simplest model is in order. Suppose you wanted to hook ten propgadgets together so that when the user drags one, the others automatically

move to keep the same value of PGA_TOP. The scheme for doing this is not to make the 45 direct interconnections between the prop gadgets but to use a central model.

Each propgadget has ICA_TARGET set to our model. Also, each prop gadget is represented by a unique IC hanging on the model's broadcast list. When one gadget gets input resulting in a change to PGA_TOP, it dutifully notifies the model (here we use the trivial maplist including { PGA_TOP, PGA_TOP } everywhere). The model in turn notifies all ICs on its broadcast list, which notify in turn all the prop gadgets, keeping them in sync.

As an added bonus, the ICA_TARGET and ICA_MAP attributes of the model are as yet unused (they are separate from the broadcast list), and can be used to represent the whole *group* of prop gadgets by notifying the outside world that the set of prop gadgets represented by the model has changed *its* value of PGA_TOP.

Advanced Models

The power of boopsi really becomes apparent when you use specializations (subclasses) of *modelclass*.

Models are extremely useful for calculating and tracking the abstract state of a group of gadgets, centralizing and broadcasting notification of changes to this state.

This type of programming, where the gadgets are interconnected to an abstract model and receive update broadcasts when model parameters change, is rooted in a concept from Smalltalk called *Model-View-Controller*.

In that system, the input aspects of a gadget (Controller) are divorced from the display aspects (View). Controllers send change notices to a Model, which in turn feeds back information causing a visual update of the original controller's corresponding View, as well as any other related View (other gadgets).

For good or ill, the concept of Controller and View are bound together in the concept of "Gadget". For that matter, gadgets individually contain some state information (e.g., PGA_TOP) which lends them some aspect of the Model, as well. But when forming interconnected networks of gadgets, it is strongly recommended that an abstract model be used to centralize state information and update broadcasts.

Another example:

Consider color controls, with two groups of three sliders, R, G, B, and H, S, V. The values of the sliders are intimately related.

The way to handle this is with an abstract color model which maintains attributes

corresponding to R,G,B,H,S, and V. When any single attribute is changed (such as Red, R) the dependent attributes (H, S, and V) are all recalculated to new values.

If the sliders are hooked up to this model so that the numeric attribute PGA_TOP is mapped to one of RGBHSV for each slider, then a change in one value will result in a notification broadcast of changes in three of the others attributes. Suitably mapped, each slider will hear about a corresponding change to PGA_TOP for the color attribute that it represents.

Note that one could easily substitute some other gadgets for the sliders, or even add a string gadget for numeric keyed input for each RGB,HSV value. This is without ANY change to the code which calculates the relationship between HSV and RGB.

The beauty of a model-based approach to interconnection becomes more and more apparent with experience in its use.

IDCMP and Other Notification Methods

As we just saw, ICs and models receive update notices and pass them along, perhaps after processing, to other boopsi objects. Other classes of objects may be defined to receive the same notification, but to forward the information in a completely different way.

Your application can get hooked into this interconnection scheme by using a special value for the attribute ICA_TARGET, specifically ICTARGET_IDCMP (defined in *intuition/icclass.h*). This will cause a new IDCMPUPDATE IntuiMessage to be sent to your application with the IAddress field pointing to a TagItem list. Also, if in the corresponding ICA_MAP, you specify a target attribute of ICSPECIAL_CODE, you can get the low-order 16 bits of the corresponding attribute value to appear in the IntuiMessage.Code field of the IDCMPUPDATE message.

This means in practice that an application can be notified via IDCMP exactly when change occurs to a value of interest, such as PGA_TOP. This is a big improvement over the existing two options: GADGETUP and FOLLOWMOUSE.

Classes which convert boopsi notification messages into still other communications protocols, such as ARexx messages, are also possible.

Whatever fancy replacements for ICs are created, the gadget, model, or IC issuing the notifications is blissfully unaware of the true nature of the target.

We have also worked out the design of a couple of ad hoc methods of "asynchronous" communication between gadgets, which are also invisible to the notifying object.

Gadget Groups

One of the more powerful concepts supported by boopsi is *composition* or grouping of objects into a single composite object. Defining a complete RGB-HSV slider/string gadget set as a single composite gadget is one example of this power.

Without getting into religious wars, composing objects out of member pieces often serves as a better method for defining new objects than using inheritance to extend an existing object class.

You can group a connected or unconnected set of gadgets together within a group gadget created from *groupgclass*, or you may define subclasses of *groupgclass* which initialize, connect, and group all the gadgets necessary to create encapsulated composite gadgets such as list scrollers, RGB/HSV controllers, or scrollbars with arrows.

See the suite of boopsi demos on the DevCon disks, demo1 through demo5 for progressing boopsi sophistication applied to solving the same problem of a slider with numeric gadget and arrows.

Class Documentation Standards

The final piece of general information needed by the "naive" boopsi gadget/class user is how the classes are documented. Theoretically, you should be able to learn all you need to know about using an object of a given class by looking at the documentation for that class.

There is a twist to that, though, in that classes can inherit behavior and attribute definitions and defaults from a unique *superclass*. That means that if a feature of a superclass is left unchanged by some subclass, then you might have to flip back through the documentation up the class hierarchy tree to examine what is defined for all superclasses.

This can be minimized by class implementors re-documenting *all* important parameters that are inherited from the superclass.

Documentation for a class consists of (at least):

- Definition of the methods that the class supports, beyond those defined by its superclass.

- Descriptions of the methods of the superclass that are overridden by this class, as well as a list of methods from the superclass that make sense.

- Descriptions of the attributes recognized by the class. The attribute ID symbol is listed, along with the type and meaning of the corresponding attribute value.

Each attribute's access mode must be defined. The flavor of our convention is borrowed from SunView, in which the access mode of an attribute is described by a string of the following capital letters indicating the access modes described:

I - Initialization

An attribute with this mode can be established when the object is created, by providing it to the `NewObject()` function.

S - Set

An attribute with this mode can be established after the object has been created, using either the function `SetAttrs` (`SetGadgetAttrs` for gadgets) or internal update notification. If some attribute can only handle one of these two methods, then it is declared as having 'S' access plus an explanatory comment in the docs.

G - Get

The value of an attribute with the mode can be retrieved using the `GetAttr()` function.

N - Notify

An attribute with this mode will give rise to an update notification broadcast whenever it is changed.

*** - Special**

There is some special restriction or feature of the processing of this attribute which warrants special discussion.

Any "granted data", i.e., data which are provided to the object along as attributes or some other scheme (such as the `OM_ADDMEMBER` method) which are the object's responsibility to dispose, must be clearly identified.

Class documentation for the system-defined classes will be attached to this document. Also, see the example programs which are available on the DevCon disks.

Boopsi Images

So far, the gadget and interconnection classes have been stressed. Another important set of classes implement a generalization of the Intuition Image structure. These are the subclasses of *imageclass*.

Image classes return objects which can be used as images in *all* Intuition contexts. These images can be rendered through whatever means the class implementor desires. Image classes could be define which implement the traditional bitplane images that are provided by struct `Image`, as well as those functions provided by `Border` and `IntuiText` structures. This alone would allow the combination of various rendering operations in a single list, rather than Intuition's current gross limitations restricting lists of artwork to be homogeneous: exclusively `Image`, `IntuiText`, or `Border`. Also, allowing programmers to implement their own image classes provides a powerful customization scheme to use whatever rendering they want to perform in Intuition contexts. Image classes to draw ellipses, boxes, 3D buttons, buttons with rounded endcaps, replacements for the Checkmark and AmigaIcon, and images which pick their colors appropriate for the Screen palette are all possible.

Writing an image class is considerably simpler than implementing a gadget or model class (images are not involved in user input or interconnection), and is a rewarding place to start. You can maintain a stash of bitmaps, a tmpras, or other data shared between the image objects created by your class, but you have to be careful to write re-entrant code and use semaphores where simultaneous access to the shared data would cause problems. A good place to stash a pointer shared data is in the `cl_UserData` field of your class structure.

Using boopsi images, you can enjoy several new functions, including:

`DrawImageState(rp, image, xoffset, yoffset, state, drawinfo)`

This is an extension of the old `DrawImage()` function which allows you to pass more information that some image classes will understand and respect.

'state': This is a value defined in `<intuition/imageclass.h>` which tells the image how to draw itself, be it normal, selected, disabled, and so on.

'drawinfo': This is a pointer to a `DrawInfo` structure defined in `<intuition/screens.h>`. It specifies an array of pen numbers to be used for highlighting, shadow, text, and so on, as well as screen depth and resolution information.

`EraseImage(rastport, image, xoffset, yoffset)`

The default for this is to erase the image's rectangular area, using the new `EraseRect()` graphics call. This call will pick up the new backfill hook for the rastport's layer, if any.

Specializations of this can be implemented in an image class to erase only that portion of the image rectangle that actually contains some image visuals (such as a circular button image), but that probably won't be practical until the `EraseRect()` functionality is extended to work with mask or fill operations.

`PointInImage(point, image)`

This function returns success or failure depending on whether the image "contains" the given point. This makes it possible to centralize the information about an image that a gadget is interested in.

This way, a button gadget can work with rectangular, round, or odd shaped images, without the need to create, install, and manage a separate mask plane or know anything about the shape of the image used.

These functions can be applied to old-fashioned "user-allocated" images, with default processing as appropriate.

Not all image classes are intended for use in all situations. For example, not all image classes will define a "selected" state, which is almost a requirement for use with a gadget. Also, some images will know how to stretch themselves to accommodate included text or other "labels", some may know how to perform like an Autoknob slider. It is also reasonable that an image class makes liberal interpretation of the drawing states, even the ones planned for future use. We see these variations as healthy.

It is only critical that documentation for a particular image class specify what the exact behavior is for the various states and special operations, and what the intended use is. It is then up to the programmer (or hacker-user) to use appropriate images for any given use.

Chapter 2: Custom Gadget Implementation

Now we return to our ascent through the levels of programming sophistication that are involved in providing these new functions to the application programmer. Implementing a gadget class is fairly tricky, but the complexity can be divided between the issues of writing a boopsi class, the issues of being a simple gadget, and interconnection/notification issues.

The issues of implementing a simple gadget can be isolated by studying the implementation of a non-boopsi custom gadget, which we explore in this chapter. Together with the information in the Chapter 4, *Writing Boopsi Classes*, the techniques for producing a custom gadget class are thus documented.

A custom gadget, to review, is a gadget which is processed by code external to Intuition, code which Intuition calls when gadgets are refreshed or the user clicks the mouse in a window, for example.

The definitions required for implementing a custom gadget can be found in `<intuition/cghooks.h>` and `<intuition/gadgetclass.h>`.

Custom Gadget Hooks

Intuition dispatches to the custom gadget implementation code via a new system structure called the Hook (defined in `<utility/hooks.h>`). A custom gadget is identified by a new value of GadgetType (CUSTOMGADGET). The address of the Hook structure is installed in the unused Gadget.MutualExclude field.

Without going into great detail, a Hook provides an entry point that Intuition can call using Amiga register parameter conventions, plus enough information to conveniently transfer control to a High-Level Language (HLL) entry point.

Once the grunt work is out of the way, a custom gadget is implemented with a single main entry point (loosely called the "gadget hook routine") which processes a variety of "commands" that are passed to it and identified by a command identification parameter (a "function code" or, when thinking of boopsi, a "Method ID").

The exact parameters passed to the gadget hook routine are:

- hook - a pointer to the gadget's hook. This provides a convenient handle for "extra information" that is common to all gadgets the hook does processing for, or as a stash for information that is only needed while a gadget is active (there is a guarantee that there will only be ONE active gadget at a time, in the eyes of Intuition).

- gadget - a pointer to the gadget itself

- parameters - a pointer to a parameter packet. The first longword of every parameter packet is the MethodID. The subsequent values depend on the Method ID. The parameter packets for the various custom gadget hooks are defined as structures in `<intuition/cghooks.h>`.

All of the various parameter packets contains a pointer to a structure called a GadgetInfo. This structure contains a collection of context information useful to gadget processing, including the requester, window, and screen the gadget occupies, a pointer to a DrawInfo structure (described above) including pen number assignments and resolution information, and a Box representing the size of the "domain" that the gadget resides in, be it requester, window, or inner region of a gimmezerozero window.

Some operations can be applied to a boopsi gadget when the gadget is not attached to a window or requester, but for vanilla custom gadgets you can assume a gadgetinfo is passed.

But there is more to it than that. Sometimes, a requester can be invisible, in which case it has no layer or rastport.

Before custom gadget code can do any rendering, it normally must bid for permission to use the rastport and layer required. The function ObtainGIRPort() is passed a GadgetInfo pointer, and returns a RastPort for rendering. If the returned RastPort pointer is ever NULL, then no rendering should be done by the custom gadget code.

This simple rule also will guarantee that a full-fledged boopsi gadget won't try to render when it's not attached to a window or is otherwise off in space.

"Gadget Relativity" refers the calculation of gadget position and dimension based on values relative to the size of a window or requester. You must perform this calculation yourself (converting negative values in the gadget position and dimension fields to positive values) according to the flags GRELBOTTOM, GRELWIDTH, and so on, *if and only if you document that your custom gadget implementation supports these flags*. You will need to do this for almost every gadget hook command, although you can stash the calculation for the period of time while your gadget is the (unique) active gadget.

Since the behavior of a custom gadget consists solely of the processing done for the various commands passed to its hook, we can complete the discussion by describing each hook command in turn. See also the example program CustGad distributed on the 1990 DevCon disks (earlier versions are obsolete).

First, a list of the MethodIDs for custom gadgets, from <intuition/gadgetclass.h>

```
GM_HITTEST
GM_RENDER
GM_GOACTIVE
GM_HANDLEINPUT
GM_GOINACTIVE
```


Now we discuss each hook command (or method) in turn. Each is passed a MethodID and GadgetInfo parameters. We will mention additional parameters.

GM_HITTEST -

Parameters include a point in gadget-relative coordinates (relative to the Gadget.LeftEdge and Gadget.TopEdge field).

You should return GMR_GADGETHIT if the point counts as a "hit" on your gadget. You may assume that Intuition has already verified that the point is in your Gadget box (LeftEdge/TopEdge/Width/Height), so a simple rectangular custom gadget can always return GMR_GADGETHIT.

You should respond with GMR_GADGETHIT properly even if you don't want your gadget to go active for some reason (such as it being disabled). This is to preserve the notion of "opacity" for your gadget by swallowing clicks that would otherwise filter through to objects beneath your gadget. You will get a subsequent opportunity to decline to go active.

GM_RENDER -

Passed a RastPort, ready for use (if non-NULL) and a "redraw mode" identifier.

In this one case, the function ObtainGIRPort() need not be called (Intuition has already set up a rastport for rendering a list of gadgets).

You are to draw your gadget. Don't forget to handle gadget relativity.

The "redraw mode" has three possible values at present. You should default to GREDRAW_REDRAW in your processing:

GREDRAW_REDRAW - full refresh, as resulting from RefreshGList(). Redraw the whole thing.

GREDRAW_TOGGLE - if your gadgets, like Intuition's, support GADGHCOMP or GADGHBORDER methods of highlighting, you can perform the toggle operation when you are passed this redraw mode.

Intuition refreshes gadgets in two passes. First it draws all the gadgets in their normal state, then it goes back and toggles the highlighting of SELECTED gadgets. This has an advantage when gadgets overlap, or when GADGHBOX highlighting is applied to abutting gadgets (the highlight box is not overwritten by adjacent gadgets), but the experienced Intuition programmer knows the downside of toggle highlighting of this nature.

We recommend that you try to draw once, in response to GREDRAW_REDRAW, in the SELECTED state if appropriate, and that you ignore GREDRAW_TOGGLE commands. This is easier using the DrawImageState() command.

GREDRAW_UPDATE - This is not encountered by a simple custom gadget, but is provided as a mechanism to update just those parts of a gadget that need to change when some attributes change in value.

Examples include the slider within a propgadget, and the text within a string gadget.

You may ignore this value or default to GREDRAW_REDRAW for simple custom gadgets (or even simple boopsi gadgets).

GM_GOACTIVE -

Passed an input event (possibly NULL), the current gadget-relative position of the mouse, and a pointer to a termination code variable.

After you respond affirmatively to GM_HITTEST or when someone calls ActivateGadget() for your gadget, you will be asked to go active by this method.

If the input event passed is NULL, you know that some equivalent to ActivateGadget has been called. If you don't support this (as buttons should not, for example) you should refuse to go active.

You can make use of the initial input event, if non-NULL. For example, string gadgets will position the cursor at this time if the input event is for a mouse button click.

You may also pre-calculate and cache information that will remain constant for the duration of a gadget being active at this time. You are guaranteed to receive a GM_GOINACTIVE when your gadget stops being active, so you can clean up. You are also guaranteed to be the unique active gadget (unless you are doing something really tricky like implementing a gadget group, which can be active and have an active member, too).

The return values and use of the termination code are the same as for GM_HANDLEINPUT, described below.

GM_HANDLEINPUT -

Once you are the active gadget (you've seen GM_GOACTIVE and agreed to become active) you will be sent all the input Intuition wants gadgets to see in GM_HANDLEINPUT calls.

The information passed is the same as for GM_GOACTIVE: an input event (never NULL this time) and a termination code pointer. This is where you perform your gadget interaction processing. You are also always given the current gadget-relative mouse position, transformed for your use.

If you want to do some rendering (which typically doesn't happen for every mouse move or timer tick), you must use ObtainGIRPort() to get a rastport to use.

You can decide to remain active or stop being active. Intuition will feel free to decide that you should go inactive, so you might receive a GM_GOINACTIVE call before you think you are done. You should also follow conventions for aborting your active state. In particular, you should go inactive if the menu button is pressed while you are active (more on this below). If you declare that you want to stop being active, you will receive a subsequent GM_GOINACTIVE call.

The termination code pointer points to a longword variable. When you go inactive with the RELVERIFY condition asserted, Intuition will send a GADGETUP message to the application window. The value in the longword termination variable will be truncated to 16 bits and put in the Code field of the GADGETUP IntuiMessage. You should not set any of the upper-16 bits in the longword, as they are reserved for future features. The value you return from the GM_HANDLEINPUT or GM_GOACTIVE processing (normal function return, in D0) specifies whether you want to remain active, whether you want a GADGETUP message transmitted, and whether the last event should be re-used or swallowed.

In order to stay active, you return GMR_MEACTIVE (which is zero) and no other values. To terminate, you return either GMR_REUSE or GMR_NOREUSE, and

optionally, GMR_VERIFY to issue the GADGETUP message. Here are some examples:

- User lets go of the mouse button while over a Boolean "command" gadget:
return (GMR_NOREUSE | GMR_VERIFY)
- User lets go of the mouse button while NOT over active Boolean "command" gadget: return (GMR_NOREUSE)
- User presses RETURN in a string gadget: return (GMR_NOREUSE|GMR_VERIFY)
- User clicks outside of the active string gadget: return (GMR_REUSE) (note that Intuition will then process this click again, to activate other windows or gadgets)
- User presses the menu button while a string gadget is active: return (GMR_REUSE) so that your gadget stops being active and Intuition will re-process the menudown event and proceed to bring up menus.

GM_GOINACTIVE -

Passed no special information, you are hereby noticed that you are no longer going to be the active gadget. Again, be advised that Intuition may decide to "abort" your active gadget, and will send this message to you before your gadget thinks that it is done.

Chapter 3: OOP Overview and Terminology

It's now time to start talking about implementing boopsi classes. Together with the principles in the previous chapter, you will ultimately be able to implement a boopsi class of custom gadgets or other objects; ultimately a public, shared class.

We can subdivide the remaining work into two areas. One is the object-oriented basis of boopsi classes and their basic mechanisms. The other is the specific methods and conventions that the major boopsi classes support.

We address the first area in this chapter. It is here that you will learn what kind of object-oriented programming system we've really got here, and learn the basics of the implementation of familiar OOP properties such as inheritance, composition, and delegation.

Objects, Methods, Messages, Classes

While we cannot hope to provide a suitable background and introduction to comparative OOP systems, we will take a stab at defining basic concepts in intuitive terms as we encounter them. If you want to get into a religious war about the difference between an object and an instance, or between delegation and inheritance, or even to decree whether boopsi is really "object-oriented", here's the chapter to worry about.

If you'd really prefer, replace all occurrences of the term "object-oriented" with "sort of object-oriented-like" if that makes you happy.

In boopsi, a "class" is a body of code which implements the creation and processing of a data structure called an "object." A class is represented by a data structure of type `Class` *<intuition/classes.h>*, which begins with an embedded Hook structure defining the entry point to the class "dispatcher" routine. Thus, a `Class` is an extended version of a Hook, so that when the class dispatcher is called, it is passed three things: a pointer to the class, a pointer to the object in question (slight variation for creating objects), and a pointer to a parameter packet often called a "message" (after Smalltalk) which always begins with a `MethodID` identifying the operation which should be performed on the object.

The basic operation of object-oriented programming is to "send a message to an object," which means to call the dispatcher for the class the object was created from, passing it the address of the class, the address of the object, and a packet of parameters containing the `MethodID` which identifies the type of operation the "message" is supposed to invoke. This is done via the function `DoMethod()` defined in the file `classface.asm`.

Each object contains a pointer to its "true class", which means that you can blindly send a message to an object without knowing its class. When programming a class implementation, you can explicitly coerce an object to be considered as a member of a specific class, typically

the superclass of the class you are authoring. Use the function `CoerceMethod()` in *classface.asm*.

It is the class dispatcher's job to interpret the `MethodID` to perform the appropriate operation for the object. Most classes in boopsi today use a `switch()` statement or equivalent to dispatch based on `MethodID`, but you could implement a cached associative jump table if you get off on that type of thing.

Frequently, messages contain a pointer to an attribute list, described above. The ID space for `MethodID`'s and attribute lists guarantees uniqueness, so you might accidentally try to invoke a method or specify an attribute that an object (actually the object's class) doesn't understand, but it will never misinterpret an ID value intended for a different context. This also means that a class can "borrow" a concept from some unrelated class, just by assigning like meaning to the ID's defined for that other class. This allows gadgets to share some of the properties of ICs (targets and maps) without having some incestuous relationship between the gadget and IC classes.

The same message can fruitfully be sent to objects from any class that understands the `MethodID`, without the caller knowing what class an object derives from. This is because the class implementation interprets the code appropriately for its objects. This is terribly convenient, eliminating, for example, the need for different functions to create or delete every different type of object, and allowing objects to notify anonymous target objects of attribute changes. This powerful concept is probably the most important in OOP, and carries the name "polymorphism" (no, not the definition I first learned for that word, either). Boopsi handles this quite nicely.

Inheritance and Transparent Base Classes

Another grand OOP concept is "inheritance." This is the traditional OOP method of selectively specializing the implementation of an existing class by programming only the extensions or differences in behavior that distinguish the new class from the original. An example, the class "propgclass" of proportional gadgets is a subclass of "gadgetclass." The implementation of "propgclass" consists of *proportional gadget specific* behavior, while most general gadget processing is left to the implementation of "gadgetclass."

When inheritance works, it's great. It is particularly useful in managing default behavior of the objects from a class. It's a simple, elegant, powerful concept with great appeal.

However, inheritance has some big problems, and it's wise, to strongly de-emphasize it in creating new classes of gadgets, especially gadgets consisting of multiple components.

Boopsi maintains simple inheritance via a pointer in the Class structure to the superclass. Although it isn't strictly necessary, it is probably wise for all classes to be subclasses of some

existing class, insuring that the default processing for object creation and deletion is performed for all objects by the common Root class ("rootclass").

The subclass relation defines what is called the "taxonomic hierarchy", also called the "class tree." There is an important point to be understood. Sometimes a subclass is called a "child" of a class, with the superclass called the "parent class". This extends to the indirect relations "ancestor" and "descendent". While these terms have appeal, it is easy for newcomers to the topic to confuse the child/parent relationship of the class tree with the very different member/composite relationship between a group and its members.

To avoid this, we dismiss the terms parent/child and ancestor/descendent, and use the terms "is some subclass of" or "is some superclass of" to describe the partial ordering that the subclass relation imposes on all classes. A boopsi class has a unique superclass, to be sure, but we might refer to "all the superclasses" of a class to refer to all classes linked via the superclass relationship from a given class all the way up to the root class.

The mechanisms of inheritance in boopsi are these:

All classes have a unique superclass (but a class may have many subclasses)

The "instance data" composing an object from a given class is a superset of the instance data of the class's superclass.

The dispatching of methods by a class defaults to passing along the message to the superclass.

New methods may be defined in a subclass.

If a subclass overrides a method defined by a superclass, it may dispatch the message to the superclass as part of its processing.

The tools for the method processing consist of a function that forwards messages to the superclass as well as the ability to send any desired message to the superclass. It is vital to understand what this means. In boopsi, it means that dispatcher of the superclass will be invoked, and will be passed the address of the object, the address of the Class data structure for the superclass, and (frequently) the same message. Use the functions DoSuperMethod() and DSM() in *classface.asm*.

It can be a difficult problem to keep track of the "currently executing class" so that referring to "the superclass" always means the right thing, especially in a loosely-bound system like boopsi. Fortunately, the current class is always passed to the class method dispatcher, so successive superclass references can be made explicitly with respect to the currently executing class. This point is kind of hard to appreciate if you haven't coded one of these systems up from scratch. The hard part is that the superclass relationship cannot be bound at link time (since you don't always get to link the class with its superclass, which may be in ROM, for example), which is how common OOP systems seem to do it.

The default case in the dispatcher switch statements is always handled by forwarding the message to the superclass.

The layout of the instance data defined by the class and all superclasses within the object data structure is nothing special: the data fields for each class follow those from its superclass in contiguous memory.

One goal was that the instance data for a class could be extended without recompiling any subclasses, so the offset from the beginning of an object to the instance data "chunk" for some class cannot be known at compile-time or link-time. This eliminates possibility of "public" or even "semi-public" data fields in instance data. When a class is initialized, its Class structure is filled in with the offset of that class's instance data within the object and the size of that data (for determining the offset of the instance data of any subclass). Thus, the instance data for each subclass follows the data for the superclass, but class methods must derive the offset of the associated instance data at run time. A fast, convenient macro is provided to add the base address of the object with the instance offset of the class.

A message sent to an object starts off by executing the dispatcher of the object's "true class"; the class it was created from. Each object contains a pointer to the object's true class, for just this purpose. In the processing, the message may "percolate up" to be processed by the superclass, and further superclasses in turn.

Method processing for an object thus *always* is running the code for "some superclass" of an object (or its true class). Instance data expected by the processing is *always* contained in the object at the offset defined for the class. Class implementation dispatchers are *always* passed a pointer to their own representative class data structure. The "purely black box" interface between a class and the instance data of its superclass(es) is essential to reserving extensibility of the instance data as described above, but can be terribly inconvenient, but since the main purpose of boopsi is to extend the functionality of gadgets and images, which are already public data structures, all hope is not lost.

Boopsi introduces the concept (soon to be common in all meaningful programming systems ;-)) of the "transparent base class." This means that the pointer to an object which belongs to some subclass of "gadgetclass" points to an actual struct Gadget, and likewise the object pointer for a subclass of "imageclass" points to a real Image structure.

As cautioned above, one must be careful in assuming that the meaning a class ascribes to these fields is the same as one is accustomed to for traditional gadgets and images, and this places a burden on the documentation for a class. But for the performance of gadget subclasses this is vital, eliminating the need to send a message up the class tree to retrieve the top edge of a gadget or to test the gadget's SELECTED flag.

Since gadgetclass and imageclass are each direct subclasses of the root class, you can deduce that the instance data for the rootclass (which should be considered private and subject to changes) *precedes* the pointer (handle) returned from NewObject().

Private vs. Public Classes

A public class is named (the ClassID is a pointer to a null-terminated string, such as "propgclass" or "imageclass"), and attached to a list of public classes which Intuition maintains. These classes can be used by an application program, or used as superclasses for other class definitions.

The method and attribute ID's for public classes must be absolutely unique, which means that there needs to be a mechanism for allocating ID space to programmers wishing to create a public class. This mechanism must be implemented by CATS. In the meantime, you can create "private" classes, which are implemented exactly the same as public classes, with two exceptions: they are not on the public class list, and their ID's are defined from a name space which does not conflict with the public class ID space. You must link the implementation of a private class into your application, and see to it that the ID's used by various private don't have conflicts that cause problems. The examples (to appear) show how to implement a private class.

Converting a private to a public class consists of redefining all the ID values to public values, and adding the class to the public list at initialization.

Public classes which are not implemented by the system can be contained in an Exec library constructed so that the classes it contains are initialized and added to the list when the library is first opened. See the boopsi example *myclass.library*.

Expunging these libraries will be a little tricky, because you may only purge the code for a class when all objects created by that class have been disposed. In some cases, this won't necessarily follow when the library is closed, so some more conditions are involved.

Chapter 4: Writing Boopsi Classes

Now that we've discussed the general principles behind boopsi, we need to talk about how to actually implement a class. For starters, the include files that will be useful include:

<code><utility/hooks.h></code>	Definition of function callback Hook structure
<code><utility/tagitems.h></code>	Ddefinitions for useful <i>utility.library</i> TagItem processing (for attribute lists)
<code><intuition/classes.h></code>	Class definitions for class implementors
<code><intuition/classusr.h></code>	ID's of Intuition system classes are defined, Method IDs and message parameter structures for the Root class are defined.
<code><intuition/gadgetclass.h></code>	Method ID's, message structures, and attribute ID's for "gadgetclass", "propgclass", "buttonclass", and "strgclass".
<code><intuition/imageclass.h></code>	Method ID's, message structures, attribute ID's for "imageclass" and the subclasses that Intuition provides, plus some useful macros for the Image "transparent base class."

You will also need to get your hands on some examples, which are on the 1990 Atlanta DevCon disks.

The first thing a class needs to have is a Hook interface to its dispatcher. See the autodocs for *utility.library* and the include file `<utility/hooks.h>` for the definitions and the examples for Lattice and "vanilla C" implementations of the hook interface.

When designing a specific class, you must first pick a superclass, and define any additional instance data that you want to be contained in the objects you create. You should also define any new methods or attributes you want your class to support.

You need some simple code to initialize a class and its hook. The Intuition routine `MakeClass()` is the last step in this process. See the examples of a private class and copy that. When you initialize your class, you specify how large your instance data is, the superclass, and the hook interface scheme you will use.

Now all you need is to implement a dispatcher routine that does the right thing with all the `MethodID`'s it may encounter, and manage the attributes defined for your class and its superclasses.

Since all classes should be subclasses of something, with the exception of the Root class, that means that all classes you write will be subclasses--perhaps indirectly so--of the Root class. So your class will be expected to implement the Root class methods, or to defer them for processing by the superclass.

Provided below is pseudo-code for processing the Root class methods. Remember that the default case for all class dispatchers should be to pass an unrecognized message along to the superclass.

The Root class method ID's are:

OM_NEW - create a new object
OM_DISPOSE - delete an object
OM_ADDTAIL - add the object to some Exec list
OM_REMOVE - remove the object from a list
OM_ADDMEMBER - for objects which include a list of other objects, add an object to the list
OM_SET - change the attributes of the object
OM_GET - retrieve the value of one of the object's attributes
OM_UPDATE - similar to set, but sent via an Interconnection
OM_NOTIFY - issue OM_UPDATE messages to others. This message is typically sent by an object to itself to broadcast changes.

We now sketch each in turn, but see the DevCon examples for more comments.

OM_NEW - Creating a new object

This method is passed the class pointer and message like all others, but it is not passed an object (since the whole idea is to create an object). In the normal "slot" for passing an object to a method dispatcher, a pointer to the "true class" from which the object is to be created. That way, all superclasses can identify if they are the "true class" of the object being created, and the Root class can refer to the true class data structure to determine how much memory to allocate for all the instance data of the object.

- 1) Pass the message along to the superclass.

Since this is done first by all classes, the result is that the Root class will allocate data for the object (the right amount for the "true class") and that working "top down" (from the Root class to its subclasses in turn, finally the class we are implementing) the instance data for the various superclasses will be initialized. This all happens before we regain control. A newly allocated object is returned, ready for us to initialize the instance data for this class.

(With our example of prop gadgets, first the Root class allocates the entire data object and initializes the Root object instance data. Then the "gadclass" code runs and initializes all Gadget data structures. Finally, control is returned to "propgadclass" which initializes propgadget-specific instance data, perhaps including some fields which belong to the Gadget (such as Gadget.SpecialInfo).

- 2) Obtain a pointer to your instance data.

Use the INST_DATA() macro, passing it the pointer to your class and the new object.

- 3) Initialize your instance data.

You may use a combination of default values with values provided in the initial attribute list you are passed. In particular, you process all "init only" attributes. You may allocate additional memory buffers for your object, or even create other objects which are components of an object of your class.

- 4) Process your initial attribute list.
After you deal with the "init only" attributes, you pass the attribute list and object through the same processing of attributes that you'll apply to an OM_SET message.
- 5) Patch up the superclass.
If you aren't happy about the superclasses seeing and making an interpretation of certain attributes (for example, a prop gadget can't support GADGHBOX gadget highlighting), you may patch them up by sending a OM_SET message to the superclass. Alternatively, you can filter "bad" attributes out of the attribute list *before* passing the OM_NEW message to the superclass (see Chapter 6, *Advanced TagItem Processing*, below).
- 6) Return the object to the caller.

OM_DISPOSE - Deleting an object

- 1) Free any additional memory you allocated.
- 2) Dispose of any objects that you created, or that were "granted" to you.
- 3) Pass the message up to the superclass, which will eventually reach the Root class, which will free all the object instance data.

OM_ADDTAIL - Add an object to a list

- 1) Pass this message up to the superclass. The Root class instance data contains a MinNode structure and code to add that node to the list passed in the OM_ADDTAIL message.

OM_REMOVE - Remove an object to a list

- 1) Pass to the superclass. The Root class will handle it.

OM_ADDMEMBER - Add a component object to a List in your object

If you define a List structure (or MinList) in your instance data, and give meaning to this method, pass the address of the list header in an OM_ADDTAIL message sent to the new member.

If you are some subclass of a class implementing this method, such as "modelclass" or "groupgclass", pass it along to the superclass.

Pass it along to the superclass even if you don't understand it.

You can iterate the objects hanging off of a List structure by using the Intuition function NextObject().

OM_SET - Change some attribute values according to a program's wishes

- 1) Pass the message along to the superclass for processing of attributes that it understands. You might want to filter out or subsequently override some attributes that you want to handle specially.
- 2) Process the attributes that your class recognizes.

- 3) Other processing depends on the nature of the class you are implementing. For an Image, you are done. Other special processing of OM_SET for different types of things includes:

For Gadgets:

If your class is the "true class" of the object (the object was created by your class, not one of your subclasses), you should refresh your visuals if any attribute value changes dictate a visual change. It is easy to check to see if you are the true class, using the OCLASS() macro.

To refresh, you must call ObtainGIRPort(), passing it the GadgetInfo pointer (if any) provided in the OM_SET message. Only perform refresh rendering if the RastPort pointer returned is not NULL. Don't forget to call ReleaseGIRPort(). You can pass a NULL RastPort pointer to ReleaseGIRPort().

If your class is not the true class, it must be some superclass of the true class. In this case, perform no refreshing, but return non-zero if some changes dictate that you should be refreshed. This information will filter down to the true class, so it can tell that refreshing is required.

For Models and Interconnections:

If any "abstract state" attributes are changed by the attribute list received, then these changes are to be broadcast to all targets.

Changes to pre-defined IC or Model attributes should not generate notification broadcasts. The classes "icclass" and "modelclass" do not broadcast changes made by OM_SET, but do broadcast anything they receive in the OM_UPDATE message.

Because a model or IC receives an OM_SET (or OM_UPDATE) may issue forth the same, some processing must be done to avoid looping conditions. You can send the ICM_CHECKLOOP message to yourself (or your superclass) to see if this object has already issued a broadcast, in which case no further processing should be done.

If ICM_CHECKLOOP returns FALSE, you should pass the attribute list to the superclass first. This will establish the values of such attributes as ICA_TARGET, which you want the superclass to handle. You are only interested in any "state" information your class wants to maintain.

Any changes to the attributes of your superclasses which might generate a notification broadcast will do so at this time. If you are a direct subclass of "modelclass" or "icclass", you are assured that no broadcasts arise from OM_SET.

If any changes are made to your own class's attributes which have the Notify access mode, you should issue an OM_NOTIFY message to yourself, which will result in the broadcasting of OM_UPDATE messages to all targets.

For gadget groups:

What you have to do here is "delegate" the attributes to the various parties interested in them, including:

- yourself, if you define any special attributes
- your superclass, for attributes it understands
- your member gadgets, which the superclass "groupgclass" will take care of
- any model or models you maintain to give "state meaning" to your gadgets.

These operations are facilitated by routines in *utility.library* which create filtered versions of the original attribute list you are passed. Deciding which objects get delegated which attributes is tricky business. The examples provide the best explanation.

OM_GET - Retrieve an object's attribute

There are no notification broadcasts involved with this message, so it's much simpler than OM_SET. If you recognize the attribute you handle the message yourself. You might delegate it to a some component, if the attribute is intended for it.

If completely unrecognized, you should pass the message to your superclass.

OM_NOTIFY - Broadcast an OM_UPDATE message to your target(s)

This message is defined by "gadgetclass", "icclass", and "modelclass." If you are some subclass of these, you should probably pass the message blindly along to your superclass.

It takes the attribute list it is passed, "maps" it according to the ICA_MAP tagitem list, and issues an OM_UPDATE message to the target ICA_TARGET, and to all the nodes in the broadcast list of a model.

You send this message "to yourself" when you want certain changes in attributes broadcast to your target objects. This message should NEVER be generated externally.

OM_UPDATE - Internal interconnection notice of attribute changes.

For gadgets, the OM_UPDATE message and OM_SET are handled the same. They both represent attribute changes requested by an external source.

For models and ICs, however, this message means "map and broadcast these changes to the target object(s)".

If you are implementing a subclass of "icclass" or "modelclass", you should perform exactly as you do for OM_SET with one exception:

Don't pass OM_UPDATE to your superclass. It will probably blindly send it unfiltered to all targets.

What you probably want to do is issue an explicit OM_NOTIFY message with the values of any of your attributes that you change.

Clearly, the explanations of the tricky parts of OM_UPDATE and OM_SET will only be understandable with reference to examples, which are provided. Things are a lot simpler in the typical cases than they might seem. For example, most subclasses of "modelclass" or "icclass" will probably be direct subclasses of "modelclass", which eliminates the mystery of "which superclasses understand and broadcast which attributes?"

Chapter 5: Advanced Tagitem List Use

The common carrier of information in boopsi is the attribute TagItem list, as defined by `utility.library` in `<utility/tagitem.h>`. As you can see, attribute lists you encounter might need to be filtered, mapped, or cloned as part of your processing. *Utility.library* provides several utility functions for this. The most common tricky operation you will do is to clone and filter an attribute list.

Whenever you send an OM_NOTIFY message, the list you pass might be modified as part of the mapping done. For performance reasons, the list is not cloned or restored, so it really can get damaged. If you intend to use the list more than once, for example if you are delegating the attributes to various objects or broadcasting it to a set of objects, you should send a copy of your original list each time. You can use the *utility.library* `CloneTagItems(taglist)` for this purpose. Then you may perform filtering and mapping to the clone without destroying the original.

If you do this, it is a performance hit to Clone, Modify, and Free the original list for each use. You should be aware of another useful function in *utility.library*, `RefreshTagItemClones(clonelist, origlist)`, which will reinitialize the clone of a list from the original, *providing you promise that you haven't changed the original since the clone was created*. This saves the Clone/Free overhead. You should also familiarize yourself with the *utility.library* functions `FilterTagItems()` and `TagInArray()` when parsing out the attributes that you want to delegate to various component objects.

Chapter 6: Boopsi Future

Down the road, several key areas can be attacked.

The attribute-value lists, which specify the parameters of the various objects, are very suitable for a textual or compiled-data "resource description" of an object which would allow a program to use a specification of a set of gadgets, models, and images which is not compiled into the program. Unlike other "resource" descriptions for user interface systems, the fact that interconnection and grouping of gadgets is data-driven means that the resource description can specify interconnections and mapping between objects, and the application's window IDCMP. This makes the resulting suite of gadgets created from a resource description more useful.

A "layout engine" for creating columns, rows, arrays, and forms of gadgets which are algorithmically sensitive to screen resolution and font size is a goal for boopsi that will not make it into the V2.0 release. It can be partially supported by external processing for the interim, but there are a lot of open issues at this time. An interactive "interface building tool" can be written to spit out either resources or C-language attribute lists for gadgets, images, and interconnections.

It will be a challenge, writing such a tool, to represent the "Model-Gadgets" philosophy of interconnection, since models are abstract quantities, not represented by some on-screen imagery. It will also be a challenge to write a modern interface layout tool that provides algorithmic concepts of "group", "aligned column", and "of equal height" which are sensitive to changes in font and resolution. Older methods which provide only the layout which can be accomplished by dragging things around with a mouse are not going to be powerful enough to handle various fonts and resolutions. And, of course, a rich set of classes needs to be written, to fulfill the needs and dreams of Intuition programmers now and in the future.

Intuition Classes Reference
 (c) Copyright 1989, 1990 Commodore-Amiga, Inc.
 All Rights Reserved
 First Version: Jim Mackrath, January 1990
 Last Revision: Jim Mackrath, June 1990
 \$Id: class.doc,v 1.3 90/06/23 17:43:18 jimm Exp \$

This document contains references information for each public "bootstrap" class defined by Intuition Version 36.

Class:
 Class ID: Root class
 Superclass: "rootclass"
 Description: None
 Include Files: Base class for all other classes
 <intuition/classes.h>
 New Methods:
 OM_NEW - create a new object. This method is passed a pointer to the "true class" of the object in place of the "yet to be created" object. The Root class will allocate enough memory for all the instance data of the true class.
 OM_DISPOSE - delete an object. This method will free the same amount of memory as allocated by OM_NEW.
 OM_ADDTAIL - add an object to an Exec list, based on a MinNode in the private rootclass object header. You can provide any Exec list you want, and iterate the objects on the list using Intuition.library/NextObject().
 OM_REMOVE - remove an object from an Exec list
 The following methods are defined for all objects, but are no-ops in the rootclass. Implementation for specific subclasses is done to support these conventions.
 OM_ADDMEMBER - Add some object (passed to the method) to the "membership" of your object, which you define. Examples include the "broadcast list" of IC's belonging to an object from modelclass, and the composite gadgets of groupclass. This is typically implemented by sending the OM_ADDTAIL message to the member object.
 OM_REMEMBER - Remove a member object previously added by OM_ADDMEMBER. You typically send the OM_REMOVE message to the member object.
 OM_SET - Set attributes from passed attribute list.
 The Root class defines no attributes.
 M_GET - Retrieve one attribute.
 OM_UPDATE - "Be updated" of external changes. A no-op for Root class.
 OM_NOTIFY - "Notify others" of changes provided in the message. A no-op for the root class.
 Changed Methods:
 Not applicable.
 Attributes:
 None.


```

Class:
Class ID:      Image class
Superclass:    "Imageclass"
Description:    Base class for Intuition-compatible Images
Include File:   <Intuition/Imageclass.h>
New Methods:

IM_DRAW - Draw the image in the Rasterport passed in the message,
at the xoffset and in the state provided in the message.
For this simple baseclass, ignores the image state passed
and just draws itself like an old-style struct Image, depending
compatibly on image data fields, including ImageData.

IM_HITTEST - Return TRUE if the point passed is "contained" in
the image. This base Image class returns TRUE if the point
is within the old struct Image box.

IM_ERASE - Erase an image. This class will call the new graphics
library function EraseRect() for its Image box. NOTE WELL:
the validity of the box dimensions (>= 0) is not tested.

IM_MOVE - erase and redraw an image. This is intended for subclasses
capable of performing animation or smooth dragging (e.g., prop
gadget knobs). A no-op for the base Image class. Never
been used.

IM_FRAMEBOX - A no-op for this baseclass, but implemented by
Image subclasses which are "stretchable", the key example
being frameclass which puts an embossed frame around
a text label or glyph. The message passed to this method
contains a pointer to a box describing the dimensions of the
contents that is to be framed, and a pointer to a "frame box"
for the result. There is also a "frameflags" field which
today has only one defined flag FRAMEF_SPECIFY.

The idea behind this frame business is to support multiple
gadgets (as found in a system requester) which share a
single frame image object, but render it in different
dimensions appropriate for their enclosed "label."

The operation of this method should be defined by suitable
subclasses as follows, and return non-zero to indicate
that they support this method.

If FRAMEF_SPECIFY is not set, you should set up the
provided frame box with the position and dimensions of the
frame you would render to enclose the contents box. This
is an inquiry function used, for example, by a gadget which
wants to know what sized frame dimensions to pass to
IM_DRAWFRAME defined below, and how to center the contents.

The caller might then add a little aesthetic margin by increasing
the dimensions (or might decide to make a column of framed
objects all the same width as the largest one), and having
done so, can call the method IM_FRAMEBOX again, this time
with FRAMEF_SPECIFY set. Now, your method is expected to
respect the dimensions specified in the frame box (even if you
think they are too small!) and to set up only the position of the
frame box appropriately for enclosing the provided contents
box. Currently implemented frame image classes always center
the contents, but future classes might want to support an
attribute for optional left or right justification of the contents.

Thankfully, there is an example provided on the Devcon disks.

```

```

IM_DRAWFRAME - Image subclasses which assign proper meaning to
this class (e.g., frameimageclass) should respect the
dimensions that are passed in this message by superceding
this method. If this baseclass sees this message, it
will convert it to an IM_DRAW message and send it back
to the image's "true class". This way, any subclass of
Imageclass which doesn't implement or recognize the
IM_DRAWFRAME method will default to drawing themselves
at their "natural" dimensions.

IM_HITFRAME - Perform a hit test as above for the image respecting
the frame dimensions passed in the message. Useful for
something like a stretchable rounded box. This class
just performs the same as IM_HITTEST, ignoring the
dimensions passed.

IM_ERASEFRAME - Erase an image respecting the frame dimensions passed
in the message. This class calls EraseRect() for the frame
dimensions, again not checking for validity.

Changed Methods:
OM_NZW - Instance data contains an Image structure, and its Depth
field is initialized to CUSTOMIMAGEDPTH, which identifies
such images to Intuition. The width and height fields are
set to arbitrary positive numbers for safety, but you should
always establish them to something meaningful.

OM_SET - Applies all supported attributes, returns '1'.

Attributes:
IA_LEFT - (ISG)
IA_TOP - (ISG)
IA_WIDTH - (ISG)
IA_HEIGHT - (ISG)
These attributes are stored, by the base Image class, as values
in the familiar Image structure.

IA_FGPN - (ISG)
IA_BGPN - (ISG)
These attributes are maintained in the Planepick and PlaneOnOff
fields, which concept they generalize.

IA_DATA - (ISG)
A pointer to general image "data". This value is stored in
the ImageData field of the old Image structure.

IA_LINEMIDTH - (I)
A no-op for the base Image class. Was supposed to be
common enough for many image classes to be defined at
this top level.

IA_NIGHTLIGHTPEN - (I)
IA_SHADOWPEN - (I)
These two images are obsolete before their time. The
intended functionality is superceded in the classes
that need it by the DrawInfo structure passed to
IM_DRAW and IM_DRAWFRAME.

IA_PENS - (I)
Defined but not supported by the base class.
Pointer to UWORD pens[], terminated by -0. This
can be used to the exclusion of DrawInfo attributes
or parameters, or as an override.

IA_RESOLUTION - (I)
Another great idea superceded by DrawInfo.

```

```

Class:
Class ID:
Superclass:
Description:
Include files:
New Methods:
None, but implements some of the no-ops in Imageclass.

```

Changed Methods:

```

OM_GET - A no-op for this class, for romspace considerations.
IM_DRAW - Draws the embossed frame in its natural (image) dimensions.
Frame thickness support of IA_LINEWIDTH is currently not implemented.
by DrawInfo and supports these drawing states:
IDS_NORMAL, IDS_INACTIVENORMAL, IDS_DISABLED;
Render edges in shadowPen, shinePen, and backgroundPen for
the interior. NOTE: No ghosting is done for IDS_DISABLED.
IDS_SELECTED, IDS_INACTIVELYSELECTED;
Render edges in shadowPen, shinePen, and hfillPen, for
the interior.
IM_DRAWFRAME - Same as IM_DRAW, but draws frame to to specified
exterior dimensions.
IM_FRAMEBOX - Centers contents, adds dimension for frame thickness,
plus an interior margin the same thickness.

Attributes:
IA_RECESSED - (Is)
Specifies that the frame should be recessed into the drawing
surface (shadowPen used on upper and left edges). Default
is FALSE, a raised frame.
IA_DOUBLEEMBOS - (Is)
Two nested embossed frames for "chisled" boxes or "ridges"
Not supported.
IA_EDGESONLY - (Is)
Does not fill frame, just draws the edges.
IA_LINEWIDTH - ()
Not supported, but should be.

```

Class: System Image Class
 Class ID: "sysiclass"
 Superclass: undocumented subclass of "imageclass"
 Description: Class for system and standard application images.
 Include File: <Intuition/imageclass.h>
 New Methods: None.

Changed Methods:

OM_NEW - Executes a drawing commands list to create a fast-rendering image object. The glyph of the object is defined by the attribute SYSI Which. The SYSI DrawInfo parameter is "required" at initialization, as is IA_HEIGHT.

Image size defaults are established by SYSI_size, but some images (title bar gadget images) require IA_HEIGHT. Images are scaled to their dimensions.

We currently do not change the parameters of these images after initialization, but there seems to be some support in there. Better documentation will be provided in the future.

Attributes:

SYSI DrawInfo - (IS)

This attribute must be passed at OM_NEW and OM_SET to allow the image to be generated into a bitmap "cache".

SYSI Which - (I)

Identifies which of the system image glyphs caller wants. Constants for the values are defined in the include file.

SYSI_size - (I)

Identifies which default dimensions to use for the object. This generalizes intuition's older concept of two different system image dimensions. The system currently uses SYSI_size_MEDRES as the default, SYSI_size_HIRES for screens with title bars greater than 22 pixels, and SYSI_size_LORES for screens with fat pixels.

Class: IntuiText Image Class
 Class ID: "inttextclass"
 Superclass: "imageclass"
 Description: IntuiText equivalent with attribute override.
 Include File: <Intuition/imageclass.h>
 New Methods: None.

Changed Methods:

IM_DRAW - Draws IntuiText specified as IA_DATA overriding the attributes specified in the IntuiText structure. The mode used is JMI, the color as specified as IA_FOPEN. The position of the "inttextclass" object is added to the position of the IntuiText.

This class was defined to make new-looking AutoRequestors without changing the IntuiText structures that were passed in. Its general usefulness is questionable, since a normal image text object would probably not carry the extra IntuiText baggage.

Attributes:

IA_DATA - (ISG)

Processed blindly by superclass, must be a pointer to a normal IntuiText structure.

IA_FOPEN - (ISG)

Processed by superclass, used as fopen when drawing text.

IA_LEFT, IA_TOP - (ISG)

Added to IntuiText offsets.

Class: Interconnection Class
 Class ID: "icclass"
 Superclass: "rootclass"
 Description: Base class of simple forwarding Interconnection.
 Include Files: <Intuition/icclass.h>
 New Methods:

ICM_SETOOP
 ICM_CLEARLOOP
 ICM_CHECKLOOP

All of these methods are used internally by subclasses to manage a "loop inhibition" for broadcasted messages. They increment, decrement, and return the value of that counter.

Changed Methods:

OM_SET - Sets attributes defined below returns 0.
 Note that this is NOT the same behavior as OM_NOTIFY.

OM_NOTIFY - Issues an OM_UPDATE message to the object indicated by attribute ICA_TARGET, first converting the attribute tags according to ICA_MAP as described in the accompanying boopl reference document.

The internal loop count is incremented until the OM_UPDATE method it invokes on ICA_TARGET returns. Also, it will do nothing at all if that loop count was non-zero to begin with, so preventing itself from participating in an infinite looping situation.

OM_UPDATE - Receive update notices from others. This base IC class just passes the message along, treating it exactly like an OM_NOTIFY message. Subclasses will probably redefine this method to perform a calculation, the result of which they might forward to others by issuing themselves an OM_NOTIFY message.

Unfortunately, this class 'converts' the OM_SET and OM_NOTIFY messages to an OM_NOTIFY by changing their MethodID field, and does not restore it to what it originally was.

Attributes:

ICA_TARGET - (IS)

Target object for OM_UPDATE messages. See the document.

If this attribute is given the value ICA_TARGET_IDCMP, then the notification will consist of sending an IDCMPUPDATE IntuiMessage to a window. The window is determined by the gadgetinfo structure passed around when the object is connected to gadgets.

ICA_MAP - (IS)

Attribute mapping list, as described in the document.

ICSPECIAL_CODE - (*)

This is a magic "dummy" attribute: If it occurs as a target in the ICA_MAP mapping list, and ICA_TARGET is ICA_TARGET_IDCMP, then the value of the corresponding notification attribute will be copied into the IntuiMessage.Code field of the IDCMPUPDATE message (just the lower sixteen bits of the attribute value will fit). This sometimes makes it particularly simple to process IDCMPUPDATE messages with a single item of interest.

Class: Model Class
 Class ID: "modelclass"
 Superclass: "icclass"
 Description: Provides "broadcast" Interconnection.
 Include Files: <Intuition/icclass.h>
 New Methods: None.

Changed Methods:
 OM_ADDMEMBER - Adds an object to the the Model's internally maintained "broadcast list." NOTE WELL: anything on this list will be DISPOSED when the model object is disposed.

OM_REMOVE - Removes objects added by OM_ADDMEMBER.

OM_DISPOSE - Disposes members of the broadcast list as well as itself.

OM_NOTIFY, OM_UPDATE - these behave as for "icclass", but first an OM_UPDATE message is sent to all members of the broadcast list, un-mapped. The members of the broadcast list are typically IC's, which have gadgets as their targets and appropriate mapping lists.

If you are creating a useful subclass, you will probably want to intercept OM_UPDATE, but pass OM_NOTIFY to this, your superclass.

Attributes:

ICA_MAP, ICA_TARGET -

Same as for the superclass.

```

Class:
  Class ID:
  Superclass:
  Description:
  Include File:
  New Methods:
    Gadget base class
    "gadgetclass"
    "rootclass"
    Base class for Intuition compatible gadget classes
    <Intuition/gadgetclass.h>

```

```

  GM_HITTEST
  GM_RENDER
  GM_GOACTIVE
  GM_HANDLEINPUT
  GM_GOINACTIVE
  Please see the Chapter 2, Custom Gadget Implementation, in
  the document for now.

```

Changed Methods:

```

OM_NEW - Initialize transparent Gadget structure. LeftEdge
and TopEdge are set to 0. Width and Height are arbitrary
constants which you should override. Sets up GadgetType
to CUSTOMGADGET, and installs a pointer to the "gadgetclass"
data structure in the MutualExclude field. These objects
are also valid "custom gadgets."

```

```

  Links self into a NextGadget linked list if GA_PREVIOUS
  is passed to OM_NEW.

```

```

OM_UPDATE and OM_SET are used simply to change the attributes.
This "gadgetclass" does not implement any concrete gadgets, so
subclasses will do more, as outlined in Chapter 4.

```

```

OM_NOTIFY - This class will issue an "IC-like" OM_UPDATE message
to ICA_TARGET through ICA_MAP. Subclasses of "gadgetclass" are
advised to let the superclass handle this method.

```

Attributes:

```

GA_LEFT - (IS)
GA_TOP - (IS)
GA_WIDTH - (IS)
GA_HEIGHT - (IS)
  These correspond to position and dimension fields in the
  Intuition Gadget structure. Setting these clears the
  "gadget relativity" flags.

```

```

GA_RELRIGHT - (IS)
GA_RELBOTTOM - (IS)
GA_RELMIDTH - (IS)
GA_RELHEIGHT - (IS)

```

```

  These are alternative position/dimension attributes. Setting
  these stores the corresponding data in the Left/Top/Width/Height
  Gadget fields, resp., and set the corresponding flag
  GRELRIGHT, GRELBOTTOM, and so on.

```

```

GA_INTUTEXT - (IS)
GA_TEXT - (IS)
GA_LABELIMAGE - (IS)

```

```

  This copies the tl Data value blindly into Gadget.GadgetText,
  and sets the flags LABELSTRING and LABELIMAGE as appropriate
  in Gadget.Flags. GA_INTUTEXT requires that tl Data be
  an IntuitText pointer, as with old-style gadgets. GA_TEXT
  takes a pointer to a null-terminated string (UBYTE *). GA_LABELIMAGE
  takes a pointer to a (boopsi) image.

```

```

Classes which support the attributes other than GA_INTUTEXT
must document themselves appropriately, but this facility
allows us to have low-overhead text gadgets and gadgets with
iconic labels.

```

```

The gadget "Image" attribute GA_IMAGE is used, in classes
frootclass, to point to the (shared) frame image that
contains the image or the button glyph.

```

```

GA_IMAGE - (IS)
  This may be a boopsi image or a regular image. Things are
  designed so that the image may be shared between gadgets.
  The image will NOT be disposed when the gadget object is disposed.

```

```

GA_BORDER - (IS)
GA_SELECTRENDER - (IS)
GA_ID - (IS)
GA_USERDATA - (IS)
GA_SPECIALINFO - (IS)
  All of this group of attributes corresponds more or less obviously
  with fields in struct Gadget.

```

```

GA_DISABLED - (IS)
GA_GIZGADGET - (IS)
GA_SELECTED - (IS)
GA_ENDGADGET - (IS)
GA_IMMEDIATE - (IS)
GA_RELVERIFY - (IS)
GA_FOLLOWMOUSE - (IS)
GA_RIGHTBORDER - (IS)
GA_LEFTBORDER - (IS)
GA_TOPBORDER - (IS)
GA_BOTTOMBORDER - (IS)
GA_TOGGLESELECT - (IS)
GA_YSGADGET - (IS)

```

```

  This group corresponds to Boolean attributes (flags) in struct
  Gadget. This class will put the correct flag in the correct
  field. You can pass any non-zero value in the tl Data tag
  field to cause the corresponding flag to be set. Pass zero
  to clear.

```

```

GA_HIGHLIGHT - (IS)
  Sets the "GADGHIHBIT" portion of Gadget.Flags to the
  attribute value in tl_Data.

```

```

GA_SYSGTYPE - (IS)
  Sets the "system gadget type" portion of Gadget.GadgetType to
  the (masked) value in tl_Data.

```

```

ICA_TARGET - (IS)
ICA_MAP - (IS)
  These are given the same meaning as in "icclass", from which
  they are "borrowed". Beats the hell out of multiple inheritance,
  doesn't it?

```

```

Notes: there are no "Get" access attributes, to save code space and
to embrace the "transparent base class" principle. If a subclass
needs you to use the OM_GET method to access any of these attributes,
it may so document.

```

Class: Proportional gadget class
 Class ID: "propclass"
 Superclass: "gadgetclass"
 Description: Extended function proportional gadgets.
 Include file: <Intuition/gadgetclass.h>
 New Methods: None.

Changed Methods:
 All methods defined by "gadgetclass" are handled to provide compatible proportional gadget processing.

OM_SET, OM_UPDATE - Changes attributes and, if needed and if _propclass is the "true class", will update the slider visuals by updating the knob position and dimensions.

GM_HANDLEINPUT - If the knob position changes sufficiently to make a difference in PGA_TOP, will issue an OM_NOTIFY message, with attributes PGA_TOP and GA_ID. The OPUF_INTERIM flag will be set for intermediate messages issued while the mouse dragging the slider knob. Will issue a message with OPUF_INTERIM clear when done, which is the "final value".

Attributes:

GA_IMAGE - (I)
 GA_BORDER - (I)
 If you don't pass GA_IMAGE to NewObject(), the gadget will create and use an AUTOKNOB. Likewise if you pass (the "illegal" attribute) GA_BORDER, an AUTOKNOB will be used instead.

GA_HIGHLIGHT - (I)
 GA_GADGBOX highlighting is not allowed, and will be converted to GADGBOX.

GA_SPECIALINFO - (I)
 This attribute is "forced" to point to the Propinfo allocated for objects of this class.

Other "gadgetclass" attributes are passed along to the superclass.

PGA_FREEDOM - (IG)
 May be one of FREEHORIZ or FREEVERT. The default is FREEVERT.

PGA_BORDERLESS - (I)
 Means the same as Propinfo.flags BORDERLESS.

PGA_HORIZPORT
 PGA_VERTPORT
 PGA_HORIZBODY
 PGA_VERTBODY

These are defined in the include file but obsolete and will not be supported. Class "Propclass" supports gadgets that are vertical or horizontally free, but not both.

PGA_TOP - (ISGNU)
 PGA_VISIBLE - (ISU)
 PGA_TOTAL - (ISU)

These attributes are very useful replacements to Pot and Body variables. They are based on the use of proportional gadgets to control scrolling text. When scrolling 100 lines of text in a 25 line visible window, you would set PGA_TOTAL to 100, PGA_VISIBLE to 25, and watch PGA_TOP run from 0 to 75 (the top line of the last page).

All internal prop gadget values will be calculated based on these (depending on whether the gadget is FREEHORIZ or FREEVERT). "Container clicks" for page jumps will leave an overlap of one line, unless the value PGA_VISIBLE is 1, in which case the prop gadget acts as an integer numeric slider taking values from 0 to PGA_TOTAL - 1.

NOTE WELL that PGA_TOP has notify access. All three of these attributes have "update access", so they can be controlled via interconnections.

Class: String gadget class
 Class ID: "strgclass"
 Superclass: "gadgetclass"
 Description: Intuition compatible string gadgets.
 Include File: <Intuition/gadgetclass.h>
 New Methods:
 None.

Changed Methods:
 All methods defined by "gadgetclass" are handled to provide compatible string gadget processing.

OM_NEW - Sets up StringInfo and StringExtend structures. Will allocate a Buffer if needed and will use shared data buffers for UndoBuffer and WorkBuffer if the MaxChars is less than SC_DEFAULTMAXCHARS (128).

Default text pens are FG = 1, BG = 0.

Attributes:

CA_ID - (ISG)
 Will be included in OM_NOTIFY messages generated.

STRINGA_MaxChars - (I)
 STRINGA_Buffer - (I)
 STRINGA_UndoBuffer - (I)
 STRINGA_WorkBuffer - (I)
 Specify various buffers defined for string gadgets and "extended string gadgets." If your value of STRINGA_MaxChars is less than SC_DEFAULTMAXCHARS (128 for now), then this class can provide all these buffers for you.

Note that UndoBuffer and WorkBuffer can be shared by many separate gadgets, providing they are as large as the largest MaxChars they will encounter.

STRINGA_BufferPos - (ISU)
 STRINGA_Dispos - (ISU)
 Familiar cursor and scroll position.

STRINGA_AlignKeyMap - (IS)
 Same as StringInfo.AlignKeyMap.

STRINGA_Font - (IS)
 Font for string gadget text. Must be an OPEN (struct TextFont *).

STRINGA_Pens - (IS)
 Pen numbers, packed as two shorts into a longword, for rendering gadget text.

STRINGA_ActivePens - (IS)
 Optional pen numbers, packed as two shorts into a longword, for rendering gadget text, when the gadget is active.

STRINGA_EditHook - (I)
 Custom string gadget edit hook (documented elsewhere).

STRINGA_EditModes - (IS)
 Value taken from flags defined in sghooks.h for initial editing modes.

STRINGA_ReplaceMode - (IS)
 STRINGA_FixedFileMode - (IS)
 STRINGA_NoFilterMode - (IS)

These three are independent Boolean equivalents to the individual flags that you can set for STRINGA_EditModes.

STRINGA_Justification - (IS)
 Takes the values STRINGCENTER, STRINGRIGHT and STRINGLEFT (which is 0).

STRINGA_LongVal - (ISGNUM)
 When you specify this to a string gadget object, it means first that the gadget is now for integer entry only, and the value of the gadget takes the numeric value passed in tl_Data.
 Note that this attribute has notify and update access.

STRINGA_TextVal - (ISGNUM)
 When you specify this to a string gadget object, it means first that the gadget is now for text entry only, and the value of the gadget is copied from the string value passed as a UBYTE pointer in tl_Data.
 Note that this attribute has notify and update access.

Notification messages will be issued whenever the gadget chooses to go inactive (not when it is aborted). The OPUF_INTERIM flag is always clear.

Class:
Class ID:
Superclass:
Description:
Include File:
New Methods:

Group gadget class
"groupclass"
"gadgetclass"
Composite gadget objects
<Intuition/gadgetclass.h>
None

Changed Methods:

OM_SET - Passes most attributes to superclass (it's "gadget self"), but propagates changes in position to its members appropriately. Also, GA_WIDTH and GA_HEIGHT are calculated from the position and dimension of the membership.

OM_ADDMEMBER - a gadget is added to the group list. The processing to propagate gadget methods (for activation and input) is "extremely" complicated. Do not try to mess with this too much. Add the gadgets you want in the group right after you create it and leave them there until you are done.

NOTE WELL that all member gadgets will be deleted by OM_DISPOSE.

All gadget methods - handled through magic.

OM_DISPOSE - this class will dispose all member gadgets.

Attributes:

LAYOUT_ORIENTATION
LAYOUT_SPACING
LAYOUT_LAYOUTOBJ

These attributes, for specifying simple layout parameters and a "layout object delegate" are not implemented.

GA_WIDTH, GA_HEIGHT are calculated from the membership.

GA_LEFT, and GA_POSITION - (IS)

These are propagated magically to the membership.

GA_RIGHT, GA_HEIGHT, and so on are not supported.

Class:
Class ID:
Superclass:
Description:
Include File:
New Methods:

Button Gadget Class
"buttonclass"
"gadgetclass"
A (repeating) command button gadget.
<Intuition/gadgetclass.h>
None.

Changed Methods:

GA_Hittest - Delegates this question to its GA_IMAGE attribute.

GA_HANDLEINPUT - Will behave like a button, but continuously issues OM_NOTIFY messages for each IECLASS_TIMER event.

Flag OPUF_INTERIM will be set for all but the last notification. The notified attribute is GA_ID, with a twist: the value sent will be the "negative" of GadgetID if the pointer is not currently over the gadget image.

GA_RENDER - All rendering is passed along to the GadgetRender image. The state provided is one of:

IDS_INACTIVELYSELECTED
IDS_SELECTED
IDS_NORMAL

Currently, more work needs to be done to support the GA_DISABLED attribute.

Attributes:

GA_IMAGE - (IS)

Changing the image will cause the gadget to refresh itself.

Class:
Class ID: Framed Command Button Gadget
Superclass: "fbuttonclass"
Description: "buttonclass"
A button gadget that knows how to outline its
"label" within a shared "Frame Image".
Include File: <Intuition/zzz.h>
New Methods:
None.

Changed Methods:

OK_NEW - Will set up its dimensions depending on GA_IMAGE, including support for frame images. If GA_IMAGE understands the IM_FRAMEBOX method, dimensions are calculated to surround the "label" stained in GadgetText, which can be GA_INTUITEXT, GA_TEXT, or GA_LABELIMAGE.

OK_SET - If you change the dimensions, will adjust the contents by using IM_FRAMEBOX with FRAMEIF_SPECIFY.

GM_HITTEST - uses IM_HITFRAME.

GM_RENDER - uses IM_DRAWFRAME. First draws the "frame", then draws the "contents" or "label" described under OK_NEW.

Attributes:

GA_WIDTH, GA_HEIGHT - (Is)
If you change these, the contents will be readjusted and the gadget re-rendered.



Programming for SCSI:

The RigidDiskBlock and Kickstart 2.0 Autoboot Strategies

by Steve Beats and Bob Burns

SCSI Direct Command

The HD_SCASICMD (IO_COMMAND value 28) was introduced to provide more flexibility when accessing devices attached to SCSI controllers. It provides a mechanism for passing SCSI and SCSI-2 command blocks directly to the devices and receiving optional sense data should that command fail. It is not possible to map every available SCSI function to the standard Amiga device commands because SCSI devices provide several functions that are "non-standard" in terms of the Amiga I/O model. The HD_SCASICMD function serves to bridge this gap.

The primary function of HD_SCASICMD is to provide an interface to SCSI tape drives, although other hardware can be just as easily supported. It also allows special commands to be sent to hard drives to modify various drive parameters that are normally inaccessible or which differ from drive to drive.

To date, the principal user of HD_SCASICMD is the HDToolBox program supplied by Commodore. This program is used to partition drives and set up filesystem segments on a reserved area of the disk. HDToolBox uses the following SCSI commands during various setup scenarios:

<i>SCSI Command</i>	<i>Used by HDToolBox</i>
INQUIRY	to find out if the unit is a hard drive and find some of the disk geometry
READ CAPACITY	to find total number of blocks available
MODE_SENSE	to find more information on drive geometry
FORMAT UNIT	for initial formatting and bad block mapping
REQUEST SENSE	to handle errors
REASSIGN BLOCKS	(optional) to map out bad blocks

(

(

(

Reading and writing of the RigidDiskBlock area (described later) is performed using normal Amiga CMD_READ and CMD_WRITE commands. For the sake of compatibility, any new non-SCSI controller should support the subset of SCSI commands listed in the table above by using the HD_SCASICMD request. This will allow end users to employ the same hard disk partitioning program for all kinds of hard disk controllers. An example is the A590 hard disk controller that supports both SCSI and XT-type drives. The XT side is handled by a separate driver that can interpret this subset of SCSI commands (including READ, WRITE and TEST UNIT READY) which allows HDToolBox to use the same commands for both devices.

Checking Up on SCSI Devices

Determining if HD_SCASICMD is supported by a device driver is a simple case of sending a benign SCSI command to unit 0 of the device. It doesn't matter if a device is hooked up or not since the error returns will be sufficient to figure out what went wrong. The best command to send is a TEST UNIT READY since this doesn't alter the media or cause any I/O (except for sense data if there was a SCSI error). A valid return, or HFERR_BadStatus, HFERR_SelTimeout, HFERR_SelUnit, or HFERR_NoBoard all indicate that the HD_SCASICMD is supported.

Finding out which SCSI addresses are populated is more difficult. The sure way is to attempt to open all available units. Since there are 56 possible SCSI addresses per controller card and timeouts can take up to two seconds, this is not always a good solution. It could take almost two minutes to scan every available SCSI address if long timeouts are enabled.

SCSI unit numbers are calculated as:

$$\text{SCSI_ID} + \text{LUN} * 10$$

Thus, unit 0 would mean address 0, LUN 0, while unit 13 means address 3, LUN 1. In the overwhelming majority of cases there are no LUNs above 0 so it is sufficient to just try units 0 through 6 (7 is usually reserved for the controller). In the cases where the controller ID is variable, then addresses 0 through 7 should be tried – but watch out for HFERR_SelUnit which means that an attempt was made to open a SCSI unit at an address that is reserved for the controller card.

When all LUN addresses are to be scanned, watch out for devices that respond to every logical unit number even though they are configured as logical unit 0. The A590 and A2091 have a jumper to disable access to LUNs greater than 0 (the A3000 uses bits in the battery-backed RAM). Since other controllers may not support this, it is necessary to run other sanity checks instead.

(

(

-

(

When multiple LUNs are opened at the same SCSI address the safest thing to do is issue a RESERVE command to each of the units. If the target supports the RESERVE command, and the multiple LUNs are really the *same* unit, then only one reservation will succeed. Don't forget to issue a corresponding RELEASE command when finished. If the unit doesn't support the RESERVE command then it is possible to fill in the LUN field in the command block. Most units that open at all LUN addresses will fail with ILLEGAL FIELD IN CDB if this address does not match their actual LUN. There are some drives that will always fail if a non-zero LUN is specified in the command block, even if their LUN is greater than zero.

Once a SCSI device has been successfully opened, don't assume it's a tape or a hard drive. Always send an INQUIRY command to determine the device type followed by a TEST UNIT READY to determine the state of the media. Before attempting any access to the media, it is advisable to send a MODE SENSE command to determine the block size. A few controllers will lock up if presented with a request for 512 bytes when the block size is larger than this. Under 2.0, variable block size devices are fully supported by the filing system so it's no longer safe to assume 512-byte blocks.

HD_SCsICMD is executed in the same manner as all other Amiga device commands. It can be sent via DoIO() or SendIO() though IOF_QUICK will always be false. In the IORequest, io_Data will point at the struct SCsICmd and io_Length should be set to sizeof(struct SCsICmd). The io_Actual and io_Offset fields are not used.

The SCsICmd Structure

The following is an extract from *devices/scsidisk.h* detailing the format of struct SCsICmd. Detailed descriptions of important fields follow.

```
struct SCsICmd {
    UWORD  *scsi_Data;      /* word aligned data for SCSI Data Phase */
                          /* (optional) data need not be byte aligned */
                          /* (optional) data need not be bus accessible */
    ULONG   scsi_Length;    /* even length of Data area */
                          /* (optional) data can have odd length */
                          /* (optional) data length can be > 2**24 */
    ULONG   scsi_Actual;    /* actual Data used */
    UBYTE   *scsi_Command; /* SCSI Command (same options as scsi_Data) */
    UWORD   scsi_CmdLength; /* length of Command */
    UWORD   scsi_CmdActual; /* actual Command used */
    UBYTE   scsi_Flags;    /* includes intended data direction */
    UBYTE   scsi_Status;   /* SCSI status of command */
    UBYTE   *scsi_SenseData; /* sense data: filled if SCsIF_[OLD]AUTONSENSE */
                          /* is set and scsi_Status has CHECK CONDITION */
                          /* (bit 1) set */
    UWORD   scsi_SenseLength; /* size of scsi_SenseData, also bytes to */
                          /* request w/ SCsIF_AUTONSENSE, must be 4..255 */
    UWORD   scsi_SenseActual; /* amount actually fetched (0 means no sense) */
};
```



```

/*----- scsi_Flags -----*/
#define SCSIF_WRITE          0 /* intended data direction is out */
#define SCSIF_READ           1 /* intended data direction is in */
#define SCSIB_READ_WRITE     0 /* (the bit to test) */

#define SCSIF_NOSENSE        0 /* no automatic request sense */
#define SCSIF_AUTONSENSE     2 /* do standard extended request sense */
                                /* on check condition */
#define SCSIF_OLDAUTONSENSE  6 /* do 4 byte non-extended request */
                                /* sense on check condition */
#define SCSIB_AUTONSENSE     1 /* (the bit to test) */
#define SCSIB_OLDAUTONSENSE  2 /* (the bit to test) */

/*----- SCSI io_Error values -----*/
#define HFERR_SelfUnit       40 /* cannot issue SCSI command to self */
#define HFERR_DMA            41 /* DMA error */
#define HFERR_Phase         42 /* illegal or unexpected SCSI phase */
#define HFERR_Parity        43 /* SCSI parity error */
#define HFERR_SelTimeout    44 /* Select timed out */
#define HFERR_BadStatus     45 /* status and/or sense error */

/*----- OpenDevice io_Error values -----*/
#define HFERR_NoBoard       50 /* Open failed for non-existent board */

```

scsi_Data

This field points to the data buffer for the SCSI data phase (if any is expected). It is generally the job of the driver software to ensure that the given buffer is DMA-accessible and to drop to programmed I/O if it isn't. Unfortunately, many controllers don't do complete checks and will lose data, or put it in the wrong place if the address is outside DMA-addressable memory. This is gradually being fixed by newer driver releases. The filing system provides a stop-gap fix with the AddressMask parameter in the mountlist. However, it is safest to restrict all direct reads and writes to CHIP RAM.

scsi_Length

This is the expected length of data to be transferred. If an unknown amount of data is to be transferred from target to host, always set this up to be larger than the most data expected. Some controllers explicitly use scsi_Length as the amount of data to transfer. In the case of variable length transfers (such as MODE SENSE) this can cause the driver to lock up since it will be waiting for a command complete message when there is really more data to transfer. The A2091, A590 and A3000 drivers always do programmed I/O for data transfers of less than 256 bytes. This prevents DMA problems with odd byte lengths.

scsi_Flags

These flags contain the intended data direction for the SCSI command. It is not strictly necessary to set the data direction flag since the SCSI protocol will inform the driver which direction data transfers will be going. However, some controllers use this info to set up DMA before issuing the command. It can also be used as a sanity check in case the data phase goes the wrong way.

(

(

2

(

SCSIF_AUTONSENSE is used to make the driver perform an automatic REQUEST SENSE if the target returns CHECK CONDITION for a SCSI command. The reason for wanting this done by the driver is because of the multi-tasking nature of the Amiga. If two tasks were accessing the same drive and the first received a CHECK CONDITION the second would destroy the sense information when it sent a command. AUTONSENSE prevents the caller from having to make two I/O requests and removes this window of vulnerability.

`scsi_SenseActual`

If SCSIF_AUTONSENSE is set it is important to initialize this field to 0 before issuing a SCSICMD. The reason is that some drivers don't support AUTONSENSE so it's important to initialize this field for them. Remember, `scsi_SenseData` is only for AUTONSENSE. If a REQUEST SENSE command is sent to the drive directly then the data will be deposited in the buffer pointed to by `scsi_Data` as usual.

One thing to remember is that `HD_SCSICMD` is geared towards an initiator role so it can't be expected to perform target-like operations. You can only send commands to a device, not receive them from an initiator. There is no provision for SCSI messaging either. This is mainly due to the interactive nature of the extended messages (such as synchronous transfer requests) which have to be handled by the driver because it knows the limitations of the controller card and has to be made aware of such protocol changes.

RigidDiskBlock – Fields and Implementation

The RigidDiskBlock standard was born from the same development effort as `HD_SCSICMD` and as a result has a heavy bias towards SCSI. However, there is nothing in the RDB specification that makes it unusable for devices using other bus protocols. The XT-style disks used in the A590 also support the RDB standard.

The RDB scheme was designed to allow the automatic mounting of all partitions on a hard drive and subsequent booting from the highest priority partition even if it has a soft loaded filing system. Disks can be removed from one controller and plugged into another (supporting the RDB scheme) and will carry with it all the necessary information for mounting and booting with them.

The preferred method of creating RigidDiskBlocks is with the `HDToolBox` program supplied by Commodore. However, there is nothing to prevent other tools being written to perform the same function.

When a driver is initialized, it uses the information contained in the RDB to mount the required partitions and mark them as bootable if needed. The driver is also responsible for loading any filing systems that are required if they are not already available on the `filesystem.resource` list. File-systems are added to the resource according to `DosType` and version number.

(

(

,

(

Here is a listing of *devices/hardblocks.h* that describes all the fields in the RDB specification.

```

/*-----
*
* This file describes blocks of data that exist on a hard disk
* to describe that disk. They are not generically accessible to
* the user as they do not appear on any DOS drive. The blocks
* are tagged with a unique identifier, checksummed, and linked
* together. The root of these blocks is the RigidDiskBlock.
*
* The RigidDiskBlock must exist on the disk within the first
* RDB_LOCATION_LIMIT blocks. This inhibits the use of the zero
* cylinder in an AmigaDOS partition: although it is strictly
* possible to store the RigidDiskBlock data in the reserved
* area of a partition, this practice is discouraged since the
* reserved blocks of a partition are overwritten by "Format",
* "Install", "DiskCopy", etc. The recommended disk layout,
* then, is to use the first cylinder(s) to store all the drive
* data specified by these blocks: i.e. partition descriptions,
* file system load images, drive bad block maps, spare blocks,, etc.
*-----*/
/*
* NOTE
* optional block addresses below contain $fffffff to indicate
* a NULL address, as zero is a valid address
*/
struct RigidDiskBlock {
    ULONG   rdb_ID;                /* 4 character identifier */
    ULONG   rdb_SummedLongs;        /* size of this checksummed structure */
    LONG    rdb_ChkSum;            /* block checksum (longword sum to zero) */
    ULONG   rdb_HostID;            /* SCSI Target ID of host */
    ULONG   rdb_BlockBytes;        /* size of disk blocks */
    ULONG   rdb_Flags;            /* see below for defines */
    /* block list heads */
    ULONG   rdb_BadBlockList;      /* optional bad block list */
    ULONG   rdb_PartitionList;    /* optional first partition block */
    ULONG   rdb_FileSysHeaderList; /* optional file system header block */
    ULONG   rdb_DriveInit;        /* optional drive-specific init code */
    /* DriveInit(lun,rdb,lor): "C" stk & d0/a0/a1 */
    ULONG   rdb_Reserved1[6];     /* set to $fffffff */
    /* physical drive characteristics */
    ULONG   rdb_Cylinders;        /* number of drive cylinders */
    ULONG   rdb_Sectors;         /* sectors per track */
    ULONG   rdb_Heads;           /* number of drive heads */
    ULONG   rdb_Interleave;      /* interleave */
    ULONG   rdb_Park;            /* landing zone cylinder */
    ULONG   rdb_Reserved2[3];
    ULONG   rdb_WritePreComp;     /* starting cylinder: write precompensation */
    ULONG   rdb_ReducedWrite;     /* starting cylinder: reduced write current */
    ULONG   rdb_StepRate;        /* drive step rate */
    ULONG   rdb_Reserved3[5];
    /* logical drive characteristics */
    ULONG   rdb_RDBBlocksLo;      /* low block of range reserved for hardblocks */
    ULONG   rdb_RDBBlocksHi;      /* high block of range for these hardblocks */
    ULONG   rdb_LoCylinder;       /* low cylinder of partitionable disk area */
    ULONG   rdb_HiCylinder;       /* high cylinder of partitionable data area */
    ULONG   rdb_CylBlocks;        /* number of blocks available per cylinder */
    ULONG   rdb_AutoParkSeconds;  /* zero for no auto park */
    ULONG   rdb_Reserved4[2];
    /* drive identification */
    char    rdb_DiskVendor[8];
    char    rdb_DiskProduct[16];
    char    rdb_DiskRevision[4];
    char    rdb_ControllerVendor[8];
    char    rdb_ControllerProduct[16];
    char    rdb_ControllerRevision[4];
    ULONG   rdb_Reserved5[10];
};

```



```

#define IDNAME_RIGIDDISK      0x5244534B      /* 'RDSK' */

#define RDB_LOCATION_LIMIT    16

#define RDBFB_LAST            0      /* no disks exist to be configured after */
#define RDBFF_LAST            0x01L /* this one on this controller */
#define RDBFB_LASTLUN         1      /* no LUNs exist to be configured greater */
#define RDBFF_LASTLUN         0x02L /* than this one at this SCSI Target ID */
#define RDBFB_LASTTID         2      /* no Target IDs exist to be configured */
#define RDBFF_LASTTID         0x04L /* greater than this one on this SCSI bus */
#define RDBFB_NORESELECT      3      /* don't bother trying to perform reselection */
#define RDBFF_NORESELECT      0x08L /* when talking to this drive */
#define RDBFB_DISKID          4      /* rdb_Disk... identification valid */
#define RDBFF_DISKID          0x10L
#define RDBFB_CTRLRID         5      /* rdb_Controller... identification valid */
#define RDBFF_CTRLRID         0x20L

/*-----*/
struct BadBlockEntry {
    ULONG   bbe_BadBlock; /* block number of bad block */
    ULONG   bbe_GoodBlock; /* block number of replacement block */
};

struct BadBlockBlock {
    ULONG   bbb_ID; /* 4 character identifier */
    ULONG   bbb_SummedLongs; /* size of this checksummed structure */
    LONG    bbb_ChkSum; /* block checksum (longword sum to zero) */
    ULONG   bbb_HostID; /* SCSI Target ID of host */
    ULONG   bbb_Next; /* block number of the next BadBlockBlock */
    ULONG   bbb_Reserved;
    struct BadBlockEntry bbb_BlockPairs[61]; /* bad block entry pairs */
    /* note [61] assumes 512 byte blocks */
};

#define IDNAME_BADBLOCK 0x42414442      /* 'BADB' */

/*-----*/
struct PartitionBlock {
    ULONG   pb_ID; /* 4 character identifier */
    ULONG   pb_SummedLongs; /* size of this checksummed structure */
    LONG    pb_ChkSum; /* block checksum (longword sum to zero) */
    ULONG   pb_HostID; /* SCSI Target ID of host */
    ULONG   pb_Next; /* block number of the next PartitionBlock */
    ULONG   pb_Flags; /* see below for defines */
    ULONG   pb_Reserved1[2];
    ULONG   pb_DevFlags; /* preferred flags for OpenDevice */
    UBYTE   pb_DriveName[32]; /* preferred DOS device name: BSTR form */
    /* (not used if this name is in use) */
    ULONG   pb_Reserved2[15]; /* filler to 32 longwords */
    ULONG   pb_Environment[17]; /* environment vector for this partition */
    ULONG   pb_EReserved[15]; /* reserved for future environment vector */
};

#define IDNAME_PARTITION      0x50415254      /* 'PART' */

#define PBFB_BOOTABLE         0      /* this partition is intended to be bootable */
#define PBFF_BOOTABLE         1L /* (expected directories and files exist) */
#define PBFB_NOMOUNT          1      /* do not mount this partition (e.g. manually) */
#define PBFF_NOMOUNT          2L /* mounted, but space reserved here) */

/*-----*/

```


(

(

,

(

```

struct FileSysHeaderBlock (
    ULONG    fhb_ID;          /* 4 character identifier */
    ULONG    fhb_SummedLongs; /* size of this checksummed structure */
    LONG     fhb_ChkSum;       /* block checksum (longword sum to zero) */
    ULONG    fhb_HostID;       /* SCSI Target ID of host */
    ULONG    fhb_Next;         /* block number of next FileSysHeaderBlock */
    ULONG    fhb_Flags;        /* see below for defines */
    ULONG    fhb_Reserved1[2];
    ULONG    fhb_DosType;      /* file system description: match this with */
                                /* partition environment's DE_DOSTYPE entry */
    ULONG    fhb_Version;      /* release version of this code */
    ULONG    fhb_PatchFlags;   /* bits set for those of the following that */
                                /* need to be substituted into a standard */
                                /* device node for this file system: e.g. */
                                /* 0x180 to substitute SegList & GlobalVec */
    ULONG    fhb_Type;         /* device node type: zero */
    ULONG    fhb_Task;         /* standard dos "task" field: zero */
    ULONG    fhb_Lock;         /* not used for devices: zero */
    ULONG    fhb_Handler;      /* filename to loadseg: zero placeholder */
    ULONG    fhb_StackSize;    /* stacksize to use when starting task */
    LONG     fhb_Priority;     /* task priority when starting task */
    LONG     fhb_Startup;      /* startup msg: zero placeholder */
    LONG     fhb_SegListBlocks; /* first of linked list of LoadSegBlocks: */
                                /* note that this entry requires some */
                                /* processing before substitution */
    LONG     fhb_GlobalVec;     /* BCPL global vector when starting task */
    ULONG    fhb_Reserved2[23]; /* (those reserved by PatchFlags) */
    ULONG    fhb_Reserved3[21];
};

#define IDNAME_FILESYSHEADER 0x46534844 /* 'FSHD' */

struct LoadSegBlock (
    ULONG    lsb_ID;          /* 4 character identifier */
    ULONG    lsb_SummedLongs; /* size of this checksummed structure */
    LONG     lsb_ChkSum;       /* block checksum (longword sum to zero) */
    ULONG    lsb_HostID;       /* SCSI Target ID of host */
    ULONG    lsb_Next;         /* block number of the next LoadSegBlock */
    ULONG    lsb_LoadData[123]; /* data for "loadseg" */
                                /* note [123] assumes 512 byte blocks */
};

#define IDNAME_LOADSEG 0x4C534547 /* 'LSEG' */

#endif /* DEVICES_HARDBLOCKS_H */

```

How the Driver Uses the RDB and Partition List

The information contained in the RigidDiskBlock and subsequent Partition blocks, *et al.*, is used by a driver in the following manner.

After determining that the target device is a hard disk (using INQUIRY), the driver will scan the first RDB_LOCATION_LIMIT (16) blocks looking for a block with the RDSK identifier and a correct sum-to-zero checksum. If no RDB is found then the driver will give up and not attempt to mount any partitions for this unit. If the RDB is found then the driver looks to see if there's a partition list for this unit (rdb_PartitionList). If none, then just the rdb_Flags will be used to determine if there are any LUNs or units after this one. This is used for early termination of the search for units on bootup.

If a partition list is present, and the partition blocks have the correct ID and checksum, then for each partition block the driver does the following:

1. Check the PBFB_NOMOUNT flag. If set then this partition is just reserving space. Skip to the next partition without mounting the current one.
2. If PBFB_NOMOUNT is false, then the partition is to be mounted. The driver fetches the given drivename from pb_DriveName. This name will be of the form dh0, work, wb_2.x, etc. A check is made to see if this name already exists on eb_MountList or DOS's device list. If it does, then the name is algorithmically altered to remove duplicates. The A590, A2091 and A3000 append .n (where n is a number) unless a name ending with .n is found. In this case the name is changed to .n+1 and the search for duplicates re-tried.
3. Next the driver constructs a parameter packet for MakeDOSNode() using the (possibly altered) drivename and information about the Exec device name and unit number. MakeDOSNode() is called to create a DOS device node. It also constructs a file system startup message from the given information and fills in defaults for the ROM filing system.
4. If MakeDOSNode() succeeds then the driver checks to see if the entry is using a standard (DOS0) filing system. If not, then the routine for patching in non-standard filing systems is called (see "Alien File Systems" below).
5. Now that the DOS node has been set up and the correct filing system segment has been associated with it, the driver checks PBFB_BOOTABLE to see if this partition is marked as bootable. If the partition is not bootable, or this is not autoboot time (DiagArea == 0) then the driver simply calls AddDosNode() to enqueue the DOS device node. If the partition is bootable, then the driver constructs a bootnode and enqueues it on eb_MountList using the bootpri from the environment vector. If this bootpri is -128 then the partition is not considered bootable.

Alien File Systems

When a filing system other than the ROM filing system is to be used, the following steps take place:

1. First, open filesystem.resource in preparation for finding the filesystem segment we want. If filesystem.resource doesn't exist then create it and add it via AddResource. Under 2.0 the resource is created by the system early on in the initialization sequence. Under 1.3 it is the responsibility of the first RDB driver to create it.
2. Scan filesystem.resource looking for a filesystem that matches theDOSType and version that we want. If it exists go to step four.
3. Since the driver couldn't find the filesystem it needed, it will have to load it from the RDB area. The list of FileSysHeaderBlocks (pointed to by the RDSK block) is scanned for a filesystem of the required DosType and version. If none is found then the driver will give up and abort the mounting of the partition. If the required filesystem is found, then it is LoadSeg'ed from the LSEG blocks and added as a new entry to the filesystem.resource.

4. The SegList pointer of the found or loaded filesystem is held in the FileSysEntry structure (which is basically an environment vector for this filing system). Using the patchflags, the driver now patches the newly created environment vector (pointed to by the new DosNode) with the values in the FileSysEntry being used. This ensures that the partition will have the correct filing system set up with the correct mount variables using a shared SegList.

The eb_Mountlist will now be set up with prioritized bootnodes and maybe some non-bootable, but mounted partitions. The system bootstrap will now take over.

Amiga BootStrap

At priority -40 in the system module initialization sequence, after most other modules are initialized, appropriate expansion boards are configured. Appropriate boards will match a FindConfigDev(, -1, -1) -- these are all boards on the expansion library board list. Furthermore, they will meet all of the following conditions:

1. CDB_CONFIGME set in cd_Flags
2. ERTB_DIAGVALID set in cd_Rom er_Type
3. Diagnostic area pointer (in cd_Rom er_Reserved0c) is non-zero
4. DAC_CONFIGTIME set in da_Config
5. At least one valid resident tag within the diagnostic area, the first of which is used by InitResident() below. This resident structure was patched to be valid during the ROM diagnostic routine run when the expansion library first initialized the board.

Boards meeting all these conditions are initialized with the standard InitResident() mechanism, with a null seglist. The board initialization code can find its ConfigDev structure with the expansion library's GetCurrentBinding() function. This is an appropriate time for drivers to Enqueue() a boot node on the expansion library's eb_MountList for use by the strap module below, and clear CDB_CONFIGME so a BindDrivers command will not try to initialize the board a second time.

This module will also enqueue nodes for 3.5" trackdisk.device units. These nodes will be at the following priorities:

Priority	Drive
5	df0:
-10	df1:
-20	df2:
-30	df3:

Next, at priority -60 in the system module initialization sequence, the strap module is invoked. Nodes from the prioritized eb_MountList list are used in priority order in attempts

to boot. An item on the list is given a chance to boot via one of two different mechanisms, depending on whether it uses boot code read in off the disk (*BootBlocks*), or uses boot code provided in the device ConfigDev diagnostic area (*BootPoint*). Floppies always use the *BootBlocks*. Other entries put on the *eb_MountList* (e.g. hard disk partitions) used the *BootPoint* mechanism for 1.3, but can use either for 1.4.

The *eb_MountList* is modified before each boot attempt, and then restored and re-modified for the next attempt if the boot fails. The node associated with the current boot attempt is placed at the head of the *eb_MountList*. Nodes marked as unusable under AmigaDOS are removed from the list. Nodes are marked as unusable by setting the most significant bit of the longword *bn_DeviceNode->dn_Handler*. This is used, for example, to keep UNIX partitions off the AmigaDOS device list when booting AmigaDOS instead of UNIX.

The selection of which of the two different boot mechanisms proceeds as follows:

1. The node must be valid boot node, i.e., it must meet both of the following conditions:
 - a) *ln_Type* is *NT_BOOTNODE*
 - b) *bn_DeviceNode* is non-zero
2. The type of boot is determined by looking at the *DosEnvec* pointed to by *fssm_Environ* pointed to by the *dn_Startup* in the *bn_DeviceNode*:
 - a) if the *de_TableSize* is less than *DE_BOOTBLOCKS*, or the *de_BootBlocks* entry is zero, *BootPoint* booting is specified, otherwise
 - b) *de_BootBlocks* contains the number of blocks to read in from the beginning of the partition, and the checksum for *BootBlocks* booting.

For *BootBlocks* booting:

1. The disk device must contain valid boot blocks:
 - a) the device and unit from *dn_Startup* opens successfully,
 - b) memory is available for the $\langle de_BootBlocks \rangle * \langle de_SizeBlock \rangle * 4$ bytes of boot block code,
 - c) the device commands *CMD_CLEAR*, *TD_CHANGENUM*, and *CMD_READ* of the boot blocks execute without error,
 - d) the boot blocks start with the three characters "DOS" and pass the longword checksum (with carry wraparound), and
 - e) memory is available to construct a boot node on the *eb_MountList* to describe the floppy. If a device error is reported in 1.c., or if memory is not available for 1.b. or 1.e., a recoverable alert is presented before continuing.
2. The boot code in the boot blocks is invoked as follows:
 - a) The address of the entry point for the boot code is offset *BB_ENTRY* into the boot blocks in memory.
 - b) The boot code is invoked with the IO Request used to issue the device commands in 1.c. above in register A1, with the *io_Offset* pointing to the beginning of the partition (the origin of the boot blocks) and *SysBase* in A6.

1

2

3

3. The boot code returns with results in both D0 and A0.
 - a) Non-zero D0 indicates boot failure. The recoverable alert AN_BootError is presented before continuing.
 - b) Zero D0 indicates A0 contains a pointer to the function to complete the boot. This completion function is chained with SysBase in A6 after the strap module frees all its resources. It is usually the DOS library initialization function, from the dos.library resident tag. Return from this function is identical to return from the strap module itself.

For *BootPoint* booting:

1. The eb_MountList node must contain a valid BootPoint:
 - a) ConfigDev pointer (in ln_Name) is non-zero,
 - b) Diagnostic area pointer (in cd_Rom er_Reserved0c) is non-zero,
 - c) DAC_CONFIGTIME set in da_Config.
2. The boot routine of a valid boot node is invoked as follows:
 - a) The address of the boot routine is calculated from da_BootPoint.
 - b) The resulting boot routine is invoked with the ConfigDev pointer on the stack in C fashion: i.e., (*boot)(configDev); Moreover, register A2 will contain the address of the associated eb_MountList node.
3. Return from the boot routine indicates failure to boot.

If all entries fail to boot, the user is prompted to put a bootable disk into a floppy drive with the "strap screen." The system floppy drives are polled for new disks. When one appears, the strap screen is removed and the appropriate boot mechanism is applied as described above. The process of prompting and trying continues till a successful boot occurs.

Changes from V1.3

The following modules have changed:

- | | |
|---------|--|
| romboot | – romboot is no longer an Exec library: RomBoot() is no longer public. A module at priority -40 still exists to do the same diagnostic area configuration. Floppies are explicitly put on the eb_MountList |
| strap | – Strap presents a new strap screen (i.e., changed the hand screen). Strap can boot from any floppy. Code is fixed to boot from non-floppy at a priority higher than five. BootBlocks support for eb_MountList entries added. BootPoint eb_MountList node parameter in A2 made official. |
| dos | – DOS no longer has special code to start up non-boot floppies. |

The following include files have changed:

- | | |
|------------------------------|--|
| dos/filehandler.[hi] | – de_BootBlocks added |
| libraries/expansionbase.[hi] | – fields privatized |
| libraries/romboot_base.[hi] | – no longer exists: BootNode now in expansionbase.[hi] ♦ |



Exec Version 2.0

The system executive (Exec) remains largely unchanged in appearance for version 2.0 Kickstart. Why fix the parts that are already very good? For 2.0, Exec has changes to support new machines and memory configurations, optimizations, programmer conveniences, and more support for the new 68020, 68030 and 68040 processors.

Long Word Alignment

The 68020 and 68030 processors are object code compatible with the 68000. Unless the 68000 code depends on CPU speed for timing, or breaks a rule, it should run without change. However, there are some simple steps that you can take to ensure optimum performance with these newer chips.

The 68020 and 68030 both have full 32 bit data busses. Under ideal conditions, each memory access takes place 32 bits at a time. However, 32 bit reads are restricted by hardware to addresses that are an even multiple of 4 bytes.

```
$0000  $ffffffee
$0004  $ddddcccc
```

A 32 bit access of location \$0000 or \$0004 will occur in one step. A 32 bit access of location \$0002 will be broken up by the processor into two parts, and will take twice as long. To see what this means for your code, consider an example:

```
struct GoodStruct          struct BadStruct
{
    char *gs_Text;          {
    ULONG  gs_This;         UBYTE  bs_That;
    UBYTE  gs_That;         char *bs_Text;
    UBYTE  gs_Foo;          ULONG  bs_This;
    UWORD  gs_Fum;          UBYTE  bs_Foo;
    struct MinList gs_Things; UWORD  gs_Fum;
    }                       struct List bs_Things;
                           }
```

The C structure on the left shows a layout optimized for speed. All pointers and long objects are grouped together at the start of the structure. The two UBYTE objects are grouped together to save space and maintain 4-byte alignment.

The structure on the right shows a bad layout. By mixing the data sizes the processor will be forced into twice as much work to fetch the same data. In the bad structure the compiler is forced to pad the structure after each UBYTE, wasting space.

Note that the good structure used a MinList structure instead of a List structure. The MinList contains only the fields needed by most programs, and has the bonus of preserving alignment. Whenever including system structures, be sure to check the length (The structure reference chart in Appendix H of the Addison-Wesley *ROM Kernel Manual: Includes and Autodocs* is designed to ease just this task).

The effect of alignment is not small. Under one test of the well-known Dhrystone benchmark, a misaligned stack cost 1/5th of the execution speed -- over 1,000 dhrystones.

Exec libraries (libs:)

Library writers should be aware that the V36 MakeLibrary() call now adjusts the location of your library base to be longword aligned. The LIB_POSSIZE and LIB_NEGSIZE fields are updated appropriately. Under V1.3 your alignment depended on how many functions the library contained. Since a "struct Library" is 34 bytes long (not a 4 byte multiple), the first entry should fix up the alignment. For example:

```
struct FooBase {
    struct Library    fb_Library;
    UWORD             fb_Flags;
    ULONG             fb_SysLib;
    struct MinList    fb_FooList;
};
```

Exec Semaphores

The system functions Forbid()/Permit() and Disable()/Enable() have been vastly overused. For many applications the Semaphore mechanism is preferred. Forbid()/Permit() and Disable()/Enable lock out *all* other tasks from the system; semaphores only affect tasks that are competing for the same resources.

A good analogy is to consider a semaphore like traffic light. A "red" semaphore prevents other cars from using an intersection; a "green" semaphore means it's safe to use. Forbid(), on the other hand, immediately stops all other cars, no matter what intersection they are near. The Disable() function is even worse, it not only stops all cars, but turns off their engines. An example of how to use semaphores is shown below:

```
/*
   A simple "do nothing" example of Exec signal semaphore use.
   When the semaphore is owned by a task, attempted access by other
   tasks will block. A nesting count is maintained, so the current
   task can safely call ObtainSemaphore() on the same semaphore.
*/
#include "exec/types.h"
#include "exec/semaphores.h"

struct SignalSemaphore LockSemaphore;

void main()
{
    InitSemaphore(&LockSemaphore);

    ObtainSemaphore(&LockSemaphore);
    printf("This task now owns the semaphore.\n");
    ReleaseSemaphore(&LockSemaphore);
}
```

Exec Devices (devs:)

Our official documentation on how to write an Exec device is the ramdrive.device example from the Addison-Wesley *ROM Kernel Manual: Includes and Autodocs*. While this example is very old, it does provide an overview of the type of things Exec devices must do, and contains many helpful comments.

The DISABLE macro is vastly overused by many devices; programmers should carefully review all uses of DISABLE, and consider the alternatives. In many cases the DISABLE can be eliminated. In other cases it

can be replaced by disabling just the interrupts of your type. For example, if your device driver works on the PORTS interrupt chain, you can disable the ports interrupt with:

```
move.w #INTF_PORTS,intena(aX)
```

Other interrupts will continue to run.

Caches and Other Secret Hiding Places

All Motorola processors support some form of cache or prefetch mechanism to enhance performance. The prefetch can overlap the time it takes to execute instructions, then the cached or prefetched data can be accessed by the CPU without the delay of a regular memory access. The effect of the caches is usually, but not always, transparent to software.

Motorola Cache Capability Outline

```
68000 One word prefetch
68010 Two word prefetch
68020 256 byte instruction-only cache. direct-mapped
68030 256 byte instruction-only cache, plus a 256
      byte data cache. Both direct-mapped. On-chip MMU.
68040 4K instruction-only cache, 4K data cache with copyback.
      Both caches are four-way set associative. Separate data
      and instruction Memory Management Units.
```

Exec processor flags (ExecBase->AttnFlags)

```
/* Processors and Co-processors: */
#define AFB_68010 0      /* also set for 68020 */
#define AFB_68020 1      /* also set for 68030 */
#define AFB_68030 2
#define AFB_68040 3
#define AFB_68881 4      /* also set for 68882 */
#define AFB_68882 5

#define AFF_68010 (1L<<0)
#define AFF_68020 (1L<<1)
#define AFF_68030 (1L<<2)
#define AFF_68040 (1L<<3)
#define AFF_68881 (1L<<4)
#define AFF_68882 (1L<<5)
```

The 68000 has the simplest prefetch mechanism. The bus controller is constantly trying to pull in an "extra" word, often before the current instruction has executed. If you watch the cycle-by-cycle activity on the 68000 bus this results in seemingly chaotic intermixing of data and instruction accesses. Sometimes a prefetched instruction will be discarded because of the effect of a branch or interrupt. A discarded prefetch is the reason the short branch instruction takes 4 more cycles than you might expect.

The 68010 adds a second word of prefetch. The primary benefit is the ability of the processor to automatically lock in the prefetched instructions. Here is an example:


```

loop:      move.l (a0)+, (a1)+      ; (20 cycles)
          dbra    d0, loop          ; (10 cycles)

```

On the 68000 the total execution time for the loop is 30 cycles. On the 68010 the time drops to just 22 cycles.

The 68020 introduces the first true instruction cache. The entries are arranged as 64 long words. The address of each instruction prefetch is indexed into an array of comparison tags. If the comparison matches, the data is pulled from the cache and no external bus cycle is needed. The supervisor bit (FC2 from the function code) is stored with each cache entry. Instructions cached in the supervisor mode will not match instructions cached in the user mode.

The 68030 has the same size instruction cache as the 68020, but it is organized as 16 entries of 4 long words each. The grouping of 4 long words is to support the burst fill capability of the 68030. In addition to the instruction cache, the 68030 also has a data cache. It is critical that areas of memory containing IO registers are not cached.

Here is a map of the 16 megabyte Amiga memory space, showing what areas should be cached.

Address	Description	Data cache status
-above 24 bits-	A3000 Extended memory area	Controlled by Zorro III rules
\$F00000-\$FFFFFF	ROM & reserved space	Cache
\$E80000-\$EFFFFFF	Autoconfig I/O space	No cache
\$DFF000-\$DFFFFFF	Custom chip registers	No cache
\$C00000-\$D7FFFF	Internal expansion memory	Cache
\$A00000-\$BFFFFFF	8520 IO chips	No cache
\$200000-\$9FFFFFF	Autoconfig	Conditional
\$000000-\$1FFFFFF	Chip memory	Cache instructions only

If data in chip memory was ever cached, the system would have major problems. DMA devices like the blitter and disk hardware can write directly to memory, without involving the CPU. Whatever data was in the cache would become stale. Any DMA device driver exposes the a risk of stale cache data.

The autoconfig areas present special problems too. There are two main types of autoconfig boards. The first contains I/O chips or status registers, the second contains memory. While you want the cache enabled for the memory, enabling the cache for chip registers would cause a disaster.

Two solutions are in use. One solution places all memory boards in the lower 8 megabyte address space, and all I/O boards in the upper address space. The areas are then defined to be cachable or not, as appropriate. This works well but limits the number of I/O type slots to eight. The second solution retains the full flexibility, but at a cost. The Memory Management Unit (MMU) is used to map all areas of memory. The cache status of each "page" can be individually specified in that case.

What the Heck Is a Write Allocate Bit?

The 68030 has a bit called "Write allocate". The wording in the 68030 manual is tricky, and it may take several readings to figure out what is going on. Here is a condensed version:

```

WA=0  Write cycles that hit replace the cached data.
      Write cycles that miss do not modify the cache.

WA=1  Write cycles that hit replace the cache data.
      Write cycles that miss invalidate the current line, and
      (if possible) update the cache.

```

The problem is that the processor stores the function code bits with each cache tag. Supervisor mode access is seen differently than user mode access. For use with the Amiga, WA must always be equal to 1.

If WA=0 the following situation can corrupt the cache:

User code reads location X, setting the valid bit in the cache.

Supervisor code writes to location X. Since the function codes are part of the tag, the write misses the cache.

User code now reads location X, and gets the old data.

Things get even trickier because of a certain "feature" of the 68030 data cache. A longword aligned write that is longword aligned will allocate a `_valid_` entry in the data cache, `_EVEN IF THE HARDWARE ASSERTS CACHE INHIBIT_`. The only way past this "feature" is to use the MMU to specify cacheing on a page-by-page basis.

How to Avoid Falling Into a Cache

All the caches explained above are somewhat "dumb." Nothing tells the cache about modifications to data or instructions that might be resident in the cache. In particular, the assumption is made that the instruction stream will not be written to. This is the reason for the rule against writing self-modifying code.

An additional wrinkle is introduced by Direct Memory Access devices (DMA). DMA access can change memory, without informing the CPU. So the risk of outdated (or "stale") data in the caches is much greater in a DMA system. V2.0 Exec provides system calls that DMA device drivers can use to clear out or flush the possibly bad data from the caches.

See the description of the `CacheControl()` Exec calls.

TABLE OF CONTENTS

`exec.library/AbortIO`
`exec.library/AllocVec`
`exec.library/CacheClearE`
`exec.library/CacheClearU`
`exec.library/CacheControl`
`exec.library/ColdReboot`
`exec.library/CreateIORequest`
`exec.library/CreateMsgPort`
`exec.library/CreatePrivatePool`
`exec.library/DeleteIORequest`
`exec.library/DeleteMsgPort`
`exec.library/DeletePrivatePool`
`exec.library/FreeVec`
`exec.library/ObtainSemaphoreShared`
`exec.library/RawDoFmt`
`exec.library/Supervisor`

New exec/types.h

```
#ifndef EXEC_TYPES_H
#define EXEC_TYPES_H
/*
**      $Id: types.h,v 36.7 90/05/10 01:07:17 bryce Exp Locker: bryce $
**
**      Data typing.  Must be included before any other Amiga include.
**
**      (C) Copyright 1985,1986,1987,1988,1989 Commodore-Amiga, Inc.
**      All Rights Reserved
**/
```

```
#define INCLUDE_VERSION 36 /* Version of the include files in use. (Do not
                           use this label for OpenLibrary() calls!) */
```

```
#define GLOBAL extern      /* the declaratory use of an external */
#define IMPORT extern      /* reference to an external */
#define STATIC static      /* a local static variable */
#define REGISTER register  /* a (hopefully) register variable */
```

```
#ifndef VOID
#define VOID      void
#endif
```

```
/* WARNING: APTR was redefined for the V1.4 Includes! APTR is a */
/* 32-Bit Absolute Memory Pointer. C pointer math will not */
/* operate on APTR -- use "ULONG" instead. */
#ifndef APTR_TYPEDEF
#define APTR_TYPEDEF
typedef void      *APTR;      /* 32-bit untyped pointer */
#endif
```

New exec/types.h

```
typedef long      LONG;      /* signed 32-bit quantity */
typedef unsigned long  ULONG; /* unsigned 32-bit quantity */
typedef unsigned long  LONGBITS; /* 32 bits manipulated individually */
typedef short      WORD;     /* signed 16-bit quantity */
typedef unsigned short  UWORD; /* unsigned 16-bit quantity */
typedef unsigned short  WORDBITS; /* 16 bits manipulated individually */
typedef signed char   BYTE;   /* signed 8-bit quantity */
typedef unsigned char  UBYTE;  /* unsigned 8-bit quantity */
typedef unsigned char  BYTEBITS; /* 8 bits manipulated individually */
typedef short      RPTR;     /* signed relative pointer */
typedef unsigned char *STRPTR; /* string pointer (NULL terminated) */

/* For compatibility only: (don't use in new code) */
typedef short      SHORT;     /* signed 16-bit quantity (use WORD) */
typedef unsigned short  USHORT; /* unsigned 16-bit quantity (use UWORD) */
typedef short      COUNT;
typedef unsigned short  UCOUNT;
typedef ULONG      CPTR;

/* Types with specific semantics */
typedef float      FLOAT;
typedef double     DOUBLE;
typedef short      BOOL;
typedef unsigned char  TEXT;

#define TRUE      1
#define FALSE     0
#ifdef NULL
#define NULL      0L
#endif

#define BYTEMASK  0xFF

/* LIBRARY_VERSION is now obsolete. Please use LIBRARY_MINIMUM */
/* or code the specific minimum library version you require. */
#define LIBRARY_VERSION LIBRARY_VERSION is obsolete--see include file
#define LIBRARY_MINIMUM33 /* Lowest version supported by Commodore-Amiga */

#endif /* EXEC_TYPES_H */
```

exec.library/AbortIO

exec.library/AbortIO

NAME

AbortIO -- attempt to abort an in-progress I/O request

SYNOPSIS

AbortIO(IORquest)

AI

VOID AbortIO(struct IORquest *);

FUNCTION

Ask a device to abort a previously started IORquest. This is done by calling the device's ABORTIO vector, with your given IORquest.

AbortIO is a command the device that may or may not grant. If successful, the device will stop processing the IORquest, and reply to it earlier than it would otherwise have done.

NOTE

AbortIO() does NOT remove the IORquest from your ReplyPort, OR wait for it to complete. After an AbortIO() you must wait normally for the reply message before actually reusing the request.

If a request has already completed when AbortIO() is called, no action is taken.

EXAMPLE

```
AbortIO(timer_request);
MaltIO (timer_request);
/* Message is free to be reused */
```

INPUTS

IORquest -- pointer to an I/O request block (must have been used at least once. May be active or finished).

RESULTS

error -- Depending on the device and the state of the request, it may not be possible to abort a given I/O request. If, for some reason the device cannot abort the request, it should return an error code in DO. Not all devices support this error return.

SEE ALSO

MaltIO, DoIO, SendIO, CheckIO

exec.library/AllocVec

exec.library/AllocVec

NAME

AllocVec -- allocate memory and keep track of the size (V36)

SYNOPSIS

memoryBlock = AllocVec(bytesize, attributes)

DO

DI

void *AllocVec(ULONG, ULONG);

FUNCTION

This function works identically to AllocMem(), but tracks the size of the allocation.

See the AllocMem() documentation for details.

WARNING

The result of any memory allocation MUST be checked, and a viable error handling path taken. Any allocation may fail if memory has been filled.

SEE ALSO

FreeVec, AllocMem

exec.library/CacheClearE

exec.library/CacheClearE

NAME CacheClearE - Instruction & data cache flushing from exceptions (V36)

SYNOPSIS
CacheClearU(control)
d0

void CacheClearU(ULONG);

FUNCTION
Clear the instruction and/or data caches from an exception or interrupt.

INPUTS
A mask of values. Use:
-1 clear all caches
-CACRF_ClearI disable clearing instruction cache
-CACRF_ClearD disable clearing data cache

SEE ALSO
CacheClearE, CacheClearU, CacheControl

exec.library/CacheClearU

exec.library/CacheClearU

NAME CacheClearU - Instruction & data cache flushing from user mode (V36)

SYNOPSIS
CacheClearU(control)
d0

void CacheClearU(ULONG);

FUNCTION
Clear the instruction and/or data caches from a task or process.

INPUTS
A mask of values. Use:
-1 clear all caches
-CACRF_ClearI disable clearing instruction cache
-CACRF_ClearD disable clearing data cache

SEE ALSO
CacheClearE, CacheClearU, CacheControl

exec.library

Page 5

exec.library/CacheControl

exec.library/CacheControl

NAME

CacheControl - Instruction & data cache control (from user mode) (V36)

SYNOPSIS

oldbits = CacheControl(cachebits, cachemask)

D0 D1

ULONG CacheControl(ULONG, ULONG);

FUNCTION

This function provides global control of any instruction or data caches that may be connected to the system. All settings are global -- per task control is not provided. The list of supported settings is provided in the exec/executebase.1 include file. While the bits currently defined map directly to the Motorola processor CACHR register, this may not be true forever. Alternative cache solutions may patch into the Exec cache functions.

Typically programmers can ignore the existence of caches. Caches will be a concern if you perform any "dirty" operations like DMA, self-modifying code, or building a jump table.

This function must only be called from the task level in user mode!

INPUTS

cachebits - new values for the bits specified in cachemask.

cachemask - a mask with ones for all bits to be changed.

RESULT

oldbits - the complete prior values for all settings.

EXAMPLE

NOTES

SEE ALSO

CacheClearF, CacheClearU, exec/executebase.1

exec.library		Page 6
exec.library/ColdReboot	exec.library/ColdReboot	
NAME	ColdReboot - reboot the Amiga (V36)	
SYNOPSIS	ColdReboot ()	
	void ColdReboot(void);	
FUNCTION	Reboot the machine. All external memory and peripherals will be RESET, and the machine will start its power up diagnostics.	
	This function never returns.	
INPUT	A chaotic pile of disoriented bits.	
RESULTS	An altogether totally integrated living system.	

```

exec.library/CreateIORequest      exec.library/CreateIORequest

NAME      CreateIORequest() -- create an IORequest structure (V36)

SYNOPSIS
    IOReq = CreateIORequest( IOReplyPort, size );
    AO
    DO

    struct IORequest *CreateIORequest(struct MsgPort *, ULONG);

FUNCTION
    Allocates memory for and initializes a new IO request block
    of a user-specified number of bytes. The number of bytes
    must be at least as large as a "struct Message".

INPUTS
    IOReplyPort - Pointer to a port for replies (an initialized message
    port, as created by CreateMsgPort()). If NULL, this
    function fails.
    size - the size of the IO request to be created.

RESULT
    IOReq - A pointer to the new IORequest block, or NULL.

SEE ALSO
    DeleteIORequest, CreateMsgPort(), amiga.lib/CreateExtIO()

```

```

exec.library/CreateMsgPort      exec.library/CreateMsgPort

NAME      CreateMsgPort - Allocate and initialize a new message port (V36)

SYNOPSIS
    CreateMsgPort()

    struct MsgPort * CreateMsgPort(void);

FUNCTION
    Allocates and initializes a new message port. The message list
    of the new port will be prepared for use (via NewList). A signal
    bit will be allocated, and the port will be set to signal your
    task when a message arrives (FA_SIGNAL).

    You "must" use DeleteMsgPort() to delete ports created with
    CreateMsgPort().

RESULT
    MsgPort - A new MsgPort structure ready for use, or NULL. If out of
    memory or signals. If you wish to add this port to the public
    port list, fill in the In_Name and In_Pri fields, then call
    AddPort(). Don't forget RemovePort().

SEE ALSO
    DeleteMsgPort(), exec/AddPort(), amiga.lib/CreatePort()

```

```

exec.library/DeleteIORequest      exec.library/DeleteIORequest

NAME      DeleteIORequest() - Free a request made by CreateIORequest() (V36)

SYNOPSIS
    DeleteIORequest( IOReq );
    AO

    void DeleteIORequest(struct IORequest *);

FUNCTION
    Frees up an IO request as allocated by CreateIORequest().

INPUTS
    IOReq - A pointer to the IORequest block to be freed, or NULL.
    This function uses the mn_length field to determine how
    much memory to free.

SEE ALSO
    CreateIORequest(), amiga.lib/DeleteExtIO()

```

```

exec.library/DeleteMsgPort      exec.library/DeleteMsgPort

NAME      DeleteMsgPort - Free a message port created by CreateMsgPort (V36)

SYNOPSIS
    DeleteMsgPort(MsgPort)
    AO

    void DeleteMsgPort(struct MsgPort *);

FUNCTION
    Frees a message port created by CreateMsgPort(). All messages that
    may have been attached to this port must have already been
    replied to.

INPUTS
    MsgPort - A message port. NULL for no action.

SEE ALSO
    CreateMsgPort(), amiga.lib/DeletePort()

```


exec.library/FreeVec

exec.library/FreeVec

NAME

FreeVec -- return AllocVec() memory to the system (V36)

SYNOPSIS

FreeVec(memoryBlock)

AI

void FreeVec(void *);

FUNCTION

Free an allocation made by the AllocVec() call. The memory will be returned to the system pool from which it came.

NOTE

If a block of memory is freed twice, the system will Guru. The Alert is ANFreeVec (\$01000009). If you pass the wrong pointer, you will probably see ANMemCorrupt \$01000005. Future versions may add more sanity checks to the memory lists.

INPUTS

memoryBlock - pointer to the memory block to free

SEE ALSO

AllocVec

exec.library/ObtainSemaphoreShared

exec.library/ObtainSemaphoreShared

NAME

ObtainSemaphoreShared -- gain shared access to a semaphore (V36)

SYNOPSIS

ObtainSemaphoreShared(signalSemaphore)

AI

FUNCTION

A lock on a signal semaphore may either be exclusive, or shared. Exclusive locks are granted by the ObtainSemaphore() and AttemptSemaphore() functions. Shared locks are granted by ObtainSemaphoreShared(). Calls may be nested.

Any number of tasks may simultaneously hold a shared lock on a semaphore. Only one task may hold an exclusive lock. A typical application is a list that is often read, but only occasionally written to.

Any exclusive locker will be held off until all shared lockers release the semaphore. Likewise, if an exclusive lock is held, all potential shared lockers will block until the exclusive lock is released. All shared lockers are restarted at the same time.

EXAMPLE

```
ObtainSemaphoreShared(ss);
/* read data */
ReleaseSemaphore(ss);

ObtainSemaphore(ss);
/* modify data */
ReleaseSemaphore(ss);
```

NOTE

While this function was added for V36, the feature magically works with all older semaphore structures.

INPUT

signalSemaphore -- an initialised signal semaphore structure

RESULT

SEE ALSO

InitSemaphore(), ReleaseSemaphore()

exec.library/RawDoFmt

exec.library/RawDoFmt

NAME RawDoFmt -- format data into a character stream.

SYNOPSIS

```
RawDoFmt(formatString, DataStream, PutChProc, PutChData);
a0 a1 a2 a3
void(char *,APTR,void (*)(),APTR);
```

FUNCTION

perform "C"-language-like formatting of a data stream, outputting the result a character at a time. Where & formatting commands are found in the formatString, they will be replaced with the corresponding element in the DataStream. % must be used in the string if a % is desired in the output.

INPUTS

formatString - a "C"-language-like NULL terminated format string, with the following supported % options:

```
%[flags][width.limit][length]type
```

flags - only one allowed. '-' specifies left justification. field width. If the first character is a '0', the field will be padded with leading 0's.

- must follow the field width, if specified

limit - maximum number of characters to output from a string. (only valid for %s).

length - size of input data defaults to WORD for types d, x, and c, 'l' changes this to long (32-bit).

type - supported types are:

b - BSTR, data is 32-bit BPTR to byte count followed by a byte string, or NULL terminated byte string. A NULL BPTR is treated as an empty string.

d - decimal (Added in V36 exec)

x - hexadecimal

s - string, a 32-bit pointer to a NULL terminated byte string. In V36, a NULL pointer is treated as an empty string

c - character

DataStream - a stream of data that is interpreted according to the format string. Often this is a pointer into the task's stack.

PutChProc - the procedure to call with each character to be output, called as:\10\fp

PutChProc(Char, PutChData);

DO-0:8 A3

the procedure is called with a NULL Char at the end of the format string.

PutChData - a value that is passed through to the PutChProc procedure. This is untouched by RawDoFmt, and may be modified by the PutChProc.

EXAMPLE

```
/ Simple version of the C "sprintf" function. Assumes C-style
/ stack-based function conventions.
```

```
/ long eyecount;
```

```
/ eyecount=2;
```

```
/ sprintf(string,"%s have %ld eyes.", "Fish", eyecount);
```

```
/ would produce "Fish have 2 eyes." in the string buffer.
```

```
/ XDEF _sprintf
```

```
/ XREF _AbsBase
```

```
/ XREF _LVORawDoFmt
```

```
/ _sprintf: movem.l a2/a3/a6, -(sp)
```

```
/ move.l 4*(sp),a3
```

```
/ move.l 5*(sp),a0
```

```
/ lea.l 6*(sp),a1
```

```
/ lea.l 7*(sp),a2
```

```
/ move.l _AbsBase,a6
```

```
/ jcr _LVORawDoFmt,a6
```

```
/ movem.l (sp)+,a2/a3/a6
```

```
/ rts
```

```
/----- PutChProc function used by RawDoFmt -----
```

```
stuffChar:
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

```
move.b d0,(a3)+
```

exec.library/Supervisor

exec.library/Supervisor

NAME

Supervisor -- trap to a short supervisor mode function

SYNOPSIS

result = Supervisor(userFunc)

Rx

As

ULONG Supervisor(void *);

FUNCTION

Execute a short assembly language function in the supervisor mode of the processor. Supervisor() does not modify or save registers; the user function has full access to the register set. The user function must end with an RTE instruction.

EXAMPLE

Obtain the Exception Vector base. 68010 or greater only!

MOVECtrap: movec.l VBR,d0 ;\$4e7a,\$0801

rte

INPUTS

userFunc - A pointer to a short assembly language function ending in RTE. The function has full access to the register set.

RESULTS

result - Whatever values the userFunc left in the registers.

SEE ALSO

SuperState/UserState



A2410, High Resolution Color Graphics Card

Hardware / Software Overview

Richard Miner Linda Wilkens Richard Lu
Alex Niedzwiecki
Center for Productivity Enhancement
University of Lowell

Abstract

A high resolution color graphics card, the A2410 has been developed for the Commodore Amiga computer. This graphics card is based on a Texas Instruments graphics systems processor, the TMS34010. The card couples the graphics system processor with frame buffer and program/data memory, a palette chip and DMA circuit for high speed data transfer between the graphics card and the Amiga.

Introduction

The A2410 high resolution graphics card is a separate graphics device that sits in one of the standard Amiga 100-pin expansion slots. The graphics card couples the TI Graphics System Processor (GSP) to its own local program memory, frame buffer memory, palette chip and DMA circuit. Presented here is an overview of the main functional components of the graphics card and a description of a low level application programmer's interface for accessing these capabilities.

High Resolution Color Graphics Card

This description of the hardware is provided to give an understanding of the graphics card architecture. Most programmers will not need this information because the device level

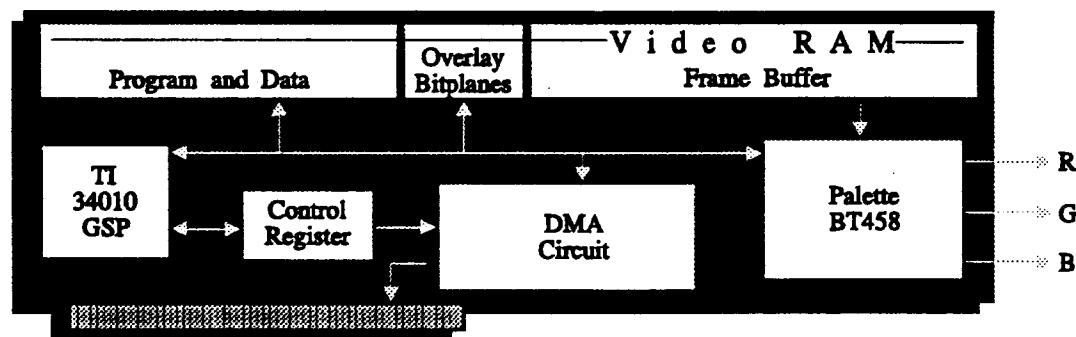


Figure 1: Block Diagram of the A2410 High Resolution Color Graphics Card

software interface to the graphics card provides a higher level abstract interface to the A2410 functionality.

The six main functional blocks of the board are depicted in Figure 1 and include:

1. Graphics System Processor (TI TMS34010)
2. Frame buffer memory
3. Program and data memory
4. Brooktree Palette chip
5. Special Register
6. DMA circuit

The Graphics System Processor

The A2410 is driven by a Texas Instruments TMS34010 *graphics system processor* (GSP). The 34010 is a general purpose CPU with an instruction set tailored for graphics applications. The GSP is responsible for communicating with the host, executing graphics instructions, refreshing memory, and updating the display. The TMS34010 is a powerful CPU which combines the features of a general-purpose processor and a graphics controller. The TMS34010 instruction set includes a full complement of general purpose instructions, as well as graphics functions, from which one can construct efficient high-level functions. The instructions support arithmetic and boolean operations, data moves, conditional jumps, subroutine calls and returns.

A2410 Memory and Register Configuration

There are two megabytes of video RAM on the A2410 split into a number of functional blocks. There is frame buffer memory for image display that supports 1024 by 1024 eight-bit pixels and additional memory for two overlay bit-planes. In addition to this image frame buffer memory there is a memory block for storing program code and data. The memory configuration for the A2410 is shown in figure 2.

In addition to the on-board memory shown in figure 2, the TMS34010 has I/O registers which are mapped to GSP addresses C0000000 through C00001F0, 15 general purpose registers called the A-file registers, 15 special purpose registers called the B-file registers, the Stack Pointer, and the Status Register. The I/O registers are used to control host interface communications, local memory interface, interrupts, video timing and screen refresh. See the TMS34010 User's Guide for a more detailed explanation. The A-file registers A0-A14 are truly general purpose, while A15 is an alias for the Stack Pointer. The B-file registers B0-B14 are used as implied operands for the 34010 graphics operations, while B15 is another alias for the Stack Pointer.

FE00 0000	+-----+
	frame
	buffer
	memory
FE7F FFFF	+-----+
FE90 0000	special reg
FF80 0000	+-----+
	overlay 0
FF90 0000	+-----+
	overlay 1
FF9F 0000	+-----+
	program
	: and :
	data memory
FFFF E000	+-----+
	traps
FFFF FFF0	+-----+

Figure 2: A2410 Hardware Memory Map

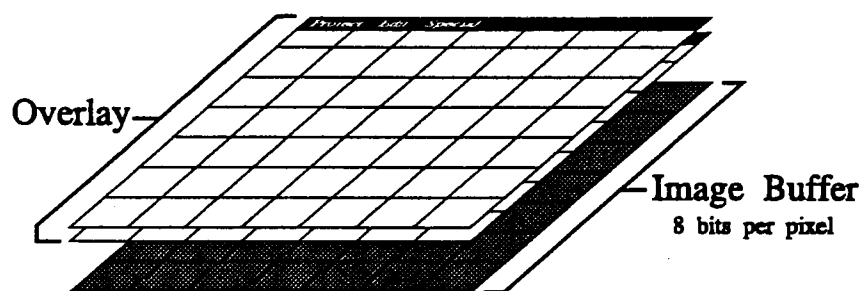


Figure 3: Frame Buffer Memory Configuration

Frame Buffer Memory

In graphics systems today, there are several major methods of representing frame buffer data and latching it through the digital to analog converters that drive displays. One method is known as *bit-plane organization*, and has separate planes of memory for each bit of every pixel in the video memory. This method is used in the native amiga graphics environment.

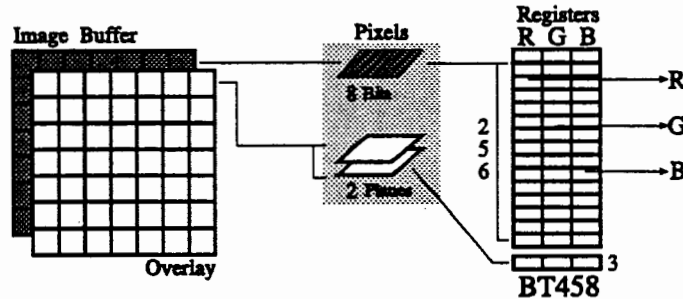


Figure 4: Color assignment through the BT458 palette chip

On the A2410 the *chunky mode* method is used. The *pixel* data is arranged contiguously in memory as consecutive eight-bit values. Additionally, there are two bit-planes of data that are used as overlay planes. The arrangement of the frame buffer and overlay planes can be seen in Figure 3.

Each eight-bit pixel in the image frame buffer is used as a pointer into a 256-element look-up table. In the look-up table the pixel value is assigned a 24-bit RGB color value. A Brooktree palette chip (BT458) provides this look-up table on the A2410. The BT458 also supports a separate look-up table that is used by the overlay bitplanes. The arrangement is shown in Figure 4.

Special Register

The A2410 Graphics Card has a number of features that utilize a special programmable register to allow the software to configure and check the status of the A2410. The A2410 Special Register, which is at GSP bit address FE90 0000, is an 8-bit register in which bits 0-3 are Read/Write, while bits 4-7 are ReadOnly. This register's bit usage is described in table 1.

TIGA

The *Texas Instruments Graphics Architecture* (TIGA) is a standard software interface for communications between host machines and graphics boards based on Texas Instruments TMS340 family of graphics processors. TIGA consists of two main pieces: the *Graphics Manager* (GM) which runs on a graphics board processor (GSP), and the *Communications Driver* (CD) which runs on the host.

Table 1: Special Register Bit Definitions

<i>Special Register Bit</i>	<i>Name</i>	<i>Description</i>
Read/Write bit 0	MOD	Display oscillator control: MOD=0: oscillator 1 MOD=1: oscillator 2
bit 1	SYNC	Composite or separate SYNC SYNC=0: separate SYNC SYNC=1: composite SYNC on Green
bit 2	SWIZZ	Byte Swap for DMA transfers SWIZZ=0: Byte swap enable SWIZZ=1: Byte swap disable
bit 3	ABR	Bus request by GSP ABR=0: no GSP bus request ABR=1: GSP bus request
Read Only bit 4	BLANK	Display Blank signal BLANK=0: blank time BLANK=1: display time
bit 5	AGBG	Bus request by Amiga AGBG=0: Amiga bus request AGBG=1: No Amiga bus request
bit 6	AID	AmigaID AID=0: A2000 AID=1: A3000
bit 7	ABG	Amiga Bus granted to GSP ABG=0: bus is granted to GSP ABG=1: bus is NOT granted to GSP

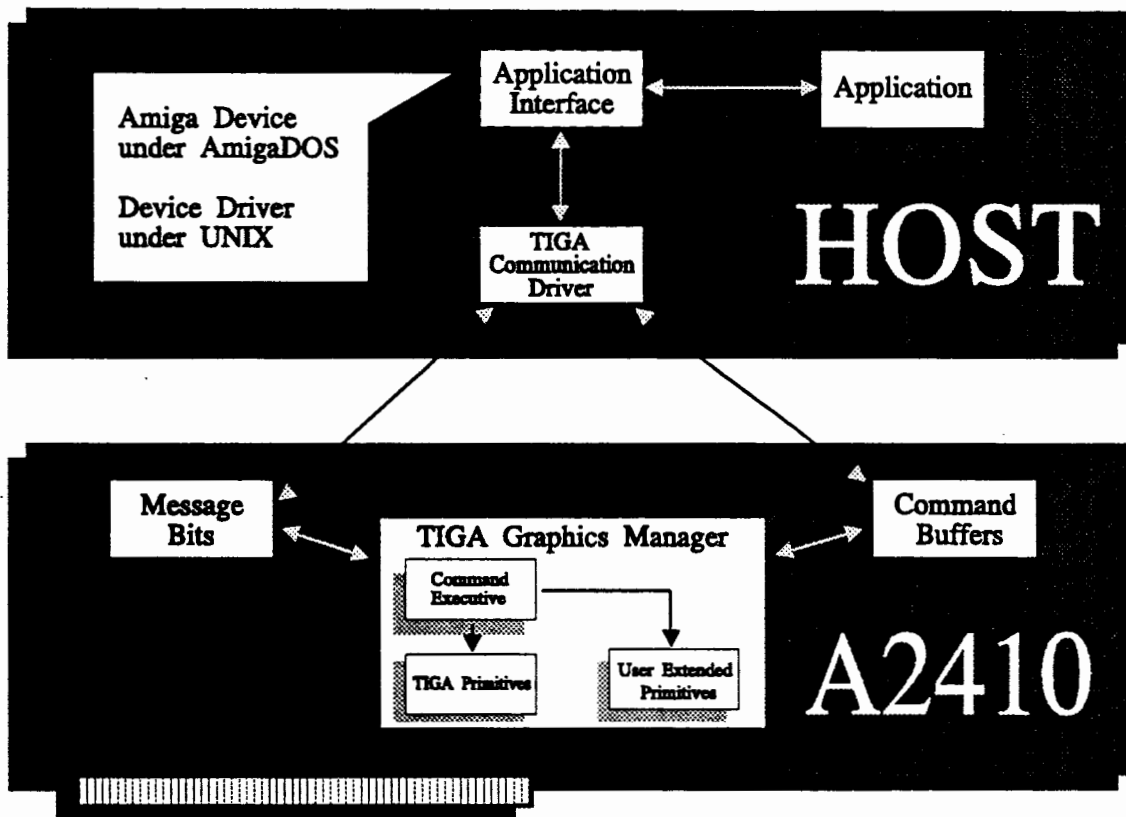


Figure 5: Texas Instruments Graphics Architecture (TIGA) Block Diagram

The CD receives requests from an application program to send a command to the graphics board and perhaps to also receive some data back from the board. The CD can be implemented in many different ways: it can be a link library, a run-time library, a device, or in an MSDOS environment, it a Terminate- and-Stay-Resident program. The GM is the only process running on the graphics processor, and hence it takes over the board completely. It stays in a main loop which waits for a command to be available and then executes it. The commands are stored in a circular buffer of command structures, with appropriate fields within the command structure being used for host to (GSP), and (GSP) to host communication.

The Graphics Manager consists of the *Command Executive* which is a set of routines for communication with the host, and up to 32 modules each of which consists of routines to perform graphics operations or to manipulate/query the current environment of the gsp. The motivation for partitioning the graphics routines up into separate modules is that different application programs may require different combinations of these modules, and the modules can be loaded individually. In reality, there are only 3 modules supplied with this release although the TIGA interface allows for customized modules to be used to supplement and/or replace two of these. Additional modules could include these with specific commands for windowing systems, image processing, advanced graphics, etc.

The supplied modules are TIGA core primitives, TIGA extended primitives, and Amiga board specific primitives. The core primitives are needed at all times. The extended prim-

Table 2: Core primitives for graphics system initialization

<i>Function</i>	<i>Description</i>
<code>function_implemented()</code>	Return if a function is implemented
<code>get_config()</code>	Return board configuration
<code>get_modeinfo()</code>	Return board mode information
<code>get_videomode()</code>	Return current mode
<code>gsp_execute()</code>	Begin execution of program on the A2410
<code>install_primitives()</code>	Install extended primitives
<code>install_usererror()</code>	Install user error handler
<code>loadcoff()</code>	Load a program onto the graphics board
<code>set_config()</code>	Set graphics configuration
<code>set_timeout()</code>	Set timeout timing value
<code>set_videomode()</code>	Set video display mode
<code>synchronize()</code>	Host waits for A2410 to execute commands
<code>clear_screen()</code>	Clears the visible portion of the screen

itives are needed to access the graphics output primitives supplied by TIGA. The Amiga board specific primitives are needed to access the special functionality resulting from the fact that the Amiga board has DMA capabilities and two overlay bitplanes.

Core primitives include functions to: initialize the graphics board, clear the screen, control graphics attributes, access the palette, access the workspace, display text, manipulate the cursor, send and receive data, implement pointer-based memory management on the graphics board, download user-defined modules, and to do some miscellaneous utility functions. Each of these categories of primitives require some additional explanation, and will be discussed in more detail. Extended primitives include functions to control a few of the graphics attributes, (except basic text display which is a core primitive), to do font selection and installation and the `get_pixel` function (since it is the inverse of the `set_pixel` function which naturally falls into the category of a graphics output primitive). The Amiga board specific primitives include DMA versions of routines to transfer data, routines to select which overlay plane to draw in, and routines to select which planes are visible.

The graphics attributes manipulated by the core primitives are foreground and background color, plane mask, pixel processing operation, transparency, and windowing. These attributes correspond to registers, or fields of a register on the graphics board which are implied operands for many of the 34010 machine level graphics instructions. The foreground color is the primary drawing color, and is usually an index into the current palette. Similarly, the background color is the secondary drawing color. The *plane mask* is a bit vector which designates which bits within a pixel are protected against writes; the 1's in the mask specify protected bits, and the 0's specify bits that can be altered. The *pixel processing operation* (ppop) determines the manner in which pixels are combined during drawing operations; for example, if ppop= 10, this implies that the destination pixel will be replaced with (source XOR destination). Transparency can be either on or off, transparency on means that if the

Table 3: Core Primitives for graphics attribute control

<i>Function</i>	<i>Description</i>
<code>cpw()</code>	Compare point to window
<code>get_colors()</code>	Return foreground and background colors
<code>get_env()</code>	Return current environment structure
<code>get_pmask()</code>	Return plane mask
<code>get_ppop()</code>	Return current pixel processing operation
<code>get_transp()</code>	Return transparency mode
<code>get_windowing()</code>	Return windowing mode
<code>set_bcolor()</code>	Set background color
<code>set_clip_rect()</code>	Set clipping rectangle
<code>set_colors()</code>	Set foreground and background colors
<code>set_fcolor()</code>	Set foreground color
<code>set_pmask()</code>	Set plane mask
<code>set_ppop()</code>	Set pixel processing operation
<code>set_windowing()</code>	Set windowing mode
<code>transp_off()</code>	Disable pixel transparency
<code>transp_on()</code>	Enable pixel transparency

Table 4: Core primitives to access the palette

<i>Function</i>	<i>Description</i>
<code>get_nearest_color()</code>	Return nearest color in the palette
<code>get_palet()</code>	Return the entire palette
<code>get_palet_entry()</code>	Return a palette entry
<code>init_palet()</code>	Initialize the default palette
<code>set_palet()</code>	Set a span of palette entries
<code>set_palet_entry()</code>	Set an individual entry in the palette

pixel value resulting from the current ppop is 0 the destination pixel is NOT altered, while transparency off means that the destination pixel is altered unconditionally. Windowing may be set at 0 for no windowing, 1 for interrupt request on write in window, 2 for interrupt request on write outside window, or 3 for clip to window. The extended drawing primitives assume windowing option 3 is set. The window to clip to is set with set_clip_rect().

Table 5: Core primitives to manipulate the workspace

<i>Function</i>	<i>Description</i>
get_wksp()	Return offscreen workspace
set_wksp()	Set the offscreen workspace

The polygon fill functions fill_polygon(), and patnfill_polygon() use an implied workspace which is the same size as the drawing area of the screen, but is only 1 bit deep. This workspace is initialized within the A2410 data memory.

Table 6: Core primitives for text

<i>Function</i>	<i>Description</i>
get_fontinfo()	Return font physical information
init_text()	Initialize text drawing environment
text_out()	Render an ASCII string

The Graphics Manager sets up a default system font, which is available for displaying text. Other fonts can be installed and used only if the extended primitives have been installed

Table 7: Core primitives to manipulate the cursor

<i>Function</i>	<i>Description</i>
get_curs_state()	Return current cursor state (OFF or ON)
get_curs_xy()	Return cursor position
set_curs_shape()	Set cursor shape
set_curs_state()	Set the cursor state
set_curs_xy()	Set current cursor position

Memory on the board is either frame buffer, or program data memory. The program memory includes program space, stack space, heap space, and the 1-bit deep workspace discussed above. Depending on the videomode currently chosen, some of the frame buffer for double buffered graphics, animation objects, etc. Thus free space available to the application is either an offscreen area, or the heap. TIGA provides dynamic heap management functions to manage the heap. The offscreen areas, if available, must be managed by the application itself.

Amiga-Board-Specific primitives The A2410 has 2 “one-bit planes” for overlays, and has DMA capabilities. The Amiga-Board-Specific (ABS) primitives are designed to access these features.

Table 8: Core primitives utility functions

<i>Function</i>	<i>Description</i>
lmo()	Return left-most-one bit number
peek_breg()	Read from a B-file register
poke_breg()	Write to a B-file register
rmo()	Return right-most-one bit number
wait_scan()	Wait for a designated scan-line

Table 9: Core primitives for memory management

<i>Function</i>	<i>Description</i>
get_offscreen_memory()	Return offscreen memory blocks
gsp_calloc()	Allocate and clear heap memory
gsp_free()	Deallocate heap memory
gsp_malloc()	Allocate memory from the heap
gsp_maxheap()	Return largest free block in the heap
gsp_minit()	Reinitialize heap
gsp_realloc()	Resize allocated block of heap memory

Table 10: Core primitives for communication

<i>Function</i>	<i>Description</i>
field_extract()	Get data from gsp memroy
field_insert()	Set data into gsp memory
get_vector()	Get address at a 34010 trap vecotr
gsp2gsp()	Copy from gsp memory to gsp memory
gsp2host()	Copy from gsp memory to host memory
gsp2hostxy()	Copy rectangular area from gsp to host
host2gsp()	Copy from host memory to gsp memory
host2gspxy()	Copy rectangular area from host to gsp
set_vector()	Set contents of gsp trap vector

Table 11: Core primitives for extensibility

<i>Function</i>	<i>Description</i>
<code>create_alm()</code>	Create absolute load module
<code>create_esym()</code>	Create external symbol table file
<code>flush_esym()</code>	Flush external symbol table file
<code>flush_extended()</code>	Flush all modules except core primitives
<code>get_isr_priorities()</code>	Return interrupt service routine priorities
<code>install_alm()</code>	Install absolute load module
<code>install_primitives()</code>	Install extended primitives
<code>install_rlm()</code>	Install relocatable load module
<code>set_interrupt()</code>	Set an interrupt handler

Table 12: Extended primitives for graphics attribute control

<i>Function</i>	<i>Description</i>
<code>set_draw_origin()</code>	Set drawing origin
<code>set_patn()</code>	Set current pattern description
<code>set_pensize()</code>	Set current pensize

The normal drawing area for graphics primitives is an section of VRAM which is treated as an x by y chunky-mode pixel array, with 8 bits per pixel. This pixel array is called plane 0. In addition, there are two overlay planes, each of which is an x by y pixel array with 1 bit per pixel. These are called plane 1 and plane 2. The ABS primitives module includes a function to select which of these 3 planes is the destination for drawing primitives, and a function to select which subset of these 3 planes are to be visible. Notice that the dimensions of the planes are controlled by the selected display mode.

There are two ways for the host and GSP to communicate. The host processor can indirectly access TMS34010 local memory by reading from or writing to the HSTDATA register, thereby accessing the word in local memory whose address is in the HSTADRL and HSTADRH registers. The CD uses this method while sending commands to the GSP and while retrieving return values from commands. The GSP can directly access Amiga memory via the A2410's DMA hardware. The TIGA command `host2gsp()`, `gsp2host()`, `host2gspxy()` and `gsp2hostxy()` hide from the user which of these transfer methods is being used.

AMIX Interface

The AMIX interface to TIGA is a link library which executes within an Amiga disk drive, all of the TIGA functions described above are available to the application program. In addition to the link library, there is a separate mechanism to load the GM before an application is executed, so that each application need not load the GM via a call to the TIGA function `loadcoff()`. The default GM contains the TIGA core primitives, the TIGA extended primitives and the Amiga-board specific primitives. An example application program to draw a

Table 13: Extended primitives for graphics output

<i>Function</i>	<i>Description</i>
<code>draw_line()</code>	Draw a line one pixel thick
<code>draw_oval()</code>	Draw an oval
<code>draw_ovalarc()</code>	Draw ellipse arc
<code>draw_piearc()</code>	Draw pie slice
<code>draw_point()</code>	Draw single pixel
<code>draw_polyline()</code>	Draw list of lines
<code>draw_rect()</code>	Draw rectangle outline
<code>fill_convex()</code>	Draw solid convex polygon
<code>fill_oval()</code>	Draw solid ellipse
<code>fill_piearc()</code>	Draw solid ellipse pie slice
<code>fill_polygon()</code>	Draw solid polygon
<code>fill_rect()</code>	Draw solid rectangle
<code>frame_oval()</code>	Draw oval border
<code>frame_rect()</code>	Draw rectangular border
<code>patnfill_convex()</code>	Draw patterned convex polygon
<code>patnfill_oval()</code>	Draw patterned ellipse
<code>patnfill_piearc()</code>	Draw patterned pieslice
<code>patnfill_polygon()</code>	Draw patterned polygon
<code>patnfill_rect()</code>	Draw patterned rectangle
<code>patnframe_oval()</code>	Draw patterned oval border
<code>patnframe_rect()</code>	Draw patterned rectangular border
<code>patnpen_line()</code>	Draw line with pattern and pen
<code>patnpen_ovalarc()</code>	Draw oval arc with pattern and pen
<code>patnpen_piearc()</code>	Draw pie slice with pattern and pen
<code>patnpen_point()</code>	Draw pixel with pattern and pen
<code>patnpen_polyline()</code>	Draw lines with pattern and pen
<code>pen_line()</code>	Draw line with pen
<code>pen_ovalarc()</code>	Draw oval arc with pen
<code>pen_piearc()</code>	Draw pie slice with pen
<code>pen_point()</code>	Draw point with pen
<code>pen_polyline()</code>	Draw lines with pen
<code>seed_fill()</code>	Fill region with foreground color
<code>seed_patnfill()</code>	Fill region with pattern
<code>styled_line()</code>	Draw styled line

Table 14: Extended primitives for pixel array functions

<i>Function</i>	<i>Description</i>
bitblt()	Bitblt source array to destination
set_dstbm()	Set destination bitmap
set_srcbm()	Set source bitmap
swap_bm()	Swap source and destination bitmaps
zoom_rect()	Zoom source rectangle

Table 15: Extended primitives for text

<i>Function</i>	<i>Description</i>
delete_font()	Remove a font from the font table
get_textattr()	Return text rendering attributes
install_font()	Install font into font table
select_font()	Select an installed font for use
set_textattr()	Set text rendering attributes
text_width()	Return the width of an ASCII string

Table 16: Extended primitives utility functions

<i>Function</i>	<i>Description</i>
get_pixel()	Read contents of a pixel

Table 17: Overlay ABS primitives

<i>Function</i>	<i>Description</i>
select_plane()	Select drawing plane
display_planes()	Choose which planes are visible

```

#include <tiga/typedefs.h>
#include <tiga/devtiga.h>
#include <tiga/tiga.h>
#include <tiga/extend.h>
#define SOURCE_PLUS_DESTINATION 16
CONFIG config;

main()
{
    short xs,ys,xs,ys,disp_hres_minus6,disp_vres_minus6;

    if (!set_videomode(TIGA,INIT | CLR_SCREEN)) exit(0);
    if (install_primitives() < 0) exit(0);

    get_config(&config);
    xs = config.mode.disp_hres>>1;
    ys = config.mode.disp_vres>>1;

    disp_hres_minus6 = config.mode.disp_hres - 6;
    disp_vres_minus6 = config.mode.disp_vres - 6;

    set_fcolor(RED);

    /* set up an add pixel processing option which will affect
    ** the overlapping lines in the center of the screen */
    set_ppop(SOURCE_PLUS_DESTINATION);

    ye = 5;                      /* draw lines at different orientations */
    for (xe = 5; xe <= disp_hres_minus6; xe += 17) draw_line(xs,ys,xs,ys);
    ye = disp_vres_minus6;
    for (xe = 5; xe <= disp_hres_minus6; xe += 17) draw_line(xs,ys,xs,ys);
    xe = 5;
    for (ye = 10; ye <= disp_vres_minus6; ye += 17) draw_line(xs,ys,xs,ys);
    xe = disp_hres_minus6;
    for (ye = 10; ye <= disp_vres_minus6; ye += 17) draw_line(xs,ys,xs,ys);

    set_videomode(PREVIOUS,INIT);
}

```

Figure 6: TIGA AMIX library interface example

```

struct TigaIOReq {
    struct Message   io_Message;
    struct Device    *io_Device; /* device private*/
    struct Unit      *io_Unit;   /* device private*/
    UWORD    Nouse;      /* not used      */
    UBYTE    io_Flags;
    BYTE     io_Error;
    ULONG    io_Actual;
    ULONG    io_Length;
    APTR     io_Data;
    ULONG    io_Offset;
    ULONG    io_Command; /* 32-bit field for tiga.device command */
    LONG     io_Return;  /* tiga.device return value */
};
\caption{TIGA Deice IO request block}

```

pattern of lines radiating from the center of the screen is shown in figure 6. This example assumes the GM is already loaded and executing on the graphics board.

AmigaDos Interface

The interface to the graphics board is implemented as an Amiga Device under AmigaDos, called "**tiga.device**". This means that the format for I/O requests, and the rules for interaction with the device task, are the same as for other Amiga devices such as the narrator device, or the serial device. Since a device unit is an instance of a device, multiple graphics boards can be supported by having only one **tiga.device**, but multiple **tiga.device** units. Advantages of implementing the interface as a device are reduced code size, language independence, and multitasking support.

Every Amiga device has an associated data structure called an "IORequest". This structure is used to direct I/O requests to the device, and if appropriate, to receive return values from the device. The **tiga.device** data structure includes the standard form of the standard IORequest structure, along with two additional fields. However, since this form uses a 16-bit command field while **tiga.device** requires a 32-bit command field, the command being sent to **tiga.device** is passed in one of its additional fields, rather than in the usual field of IOStdReq. Thus the **tiga.device** I/O request block is defined in figure 7.

As for any device, **io_Message** is a message header used by the device to queue I/O requests and to return I/O requests upon completion. The application program must set this up properly for I/O to work correctly. The **io_Device** and **io_Unit** fields are private to the device, and are not touched by the application. The next field is 'Nouse', or not used. The **io_Flags** field is divided into the lower nibble which is used by AmigaDos Exec, and the

```

1- #include <tiga/typedefs.h> /* standard TIGA include file */
2- #include <tiga/devtiga.h> /* definition of tiga.device commands */
3- LONG tiga_data[TIGA_DATA_SIZE]; /* the data buffer */
4- struct TigaIOReq tiga;          /* TIGA io request block */

5- main()
6- {

7-     /* initialize TIGA interface */
8-     tiga.io_Message.mn_ReplyPort = CreatePort(NULL,0);
9-     if (tiga.io_Message.mn_ReplyPort == NULL){
10- printf("Can't create the reply port\n");
11-     exit(0);
12- }
13-     tiga.io_Data = (APTR) tiga_data;

14-     if ((OpenDevice("tiga.device",0,&tiga,0)) != NULL) {
15-         printf("can't open tiga device\n");
16-         DeletePort(tiga.io_Message.mn_ReplyPort);
17-         exit(0);
18-     }

19-     /* initialize the graphics board */
20-     tiga.io_Command = TIGA_SET_VIDEOMODE;
21-     tiga_data[0] = (LONG)(TIGA);
22-     tiga_data[1] = (LONG)(INIT | CLR_SCREEN);
23-     DoIO(&tiga);
24-     retval = tiga.io_Return;
25-     printf("Return value from set_videomode = %lx\n",retval);

26-     /* close tiga.device */
27-     CloseDevice(&tiga);
28-     DeletePort(tiga.io_Message.mn_ReplyPort);
29- }

```

Figure 7: TIGA AmigaDOS device interface example

upper nibble which is not currently used by `tiga.device`, but which is reserved for future use. The `io_Error` field is used by the device to return an error or a warning value upon request completion. Neither the `io_Actual` nor the `io_Length` fields are currently used by `tiga.device`, but may be used in the future. The `io_Data` field is a pointer to the data buffer, the space for which must be provided by the application. The actual data buffer is treated as an array of LONG's. The `io_Offset` field is not used by `tiga.device`. The `io_Command` field must be set to a valid `tiga.device` command before the device is called. The commands recognized by `tiga.device` are derived from the TIGA functions, and are defined in the header file `devtiga.h`. The last field `io_Return` is used by `tiga.device` to return a value to the application.

The following is a description of the example code presented in figure 8. Line number 1 includes the header file which defines TIGA data types that may be referred to in an application. Line number 2 includes the header file that defines the `tiga.device` commands. In line numbers 3 and 4 we see the definitions of the data buffer and the `IORequest` structure. Notice that in line 13 these are coupled by assigning the data buffer to the appropriate field in the `IORequest` structure. Lines 7 - 18 properly prepare the `IORequest` structure, and open `tiga.device`. Lines number 20 through 24 send a request to the device to initialize the graphics board, and to clear the screen. `DoIO` will wait until the device has performed the requested command. The return value is then available in `tiga.io_Return`. Lines 27 and 28 close the device, and delete the reply port that was created in line 8.

In the above example, the assignments in lines 20-23 are could be replaced by an equivalent macro which is provided in the header file `dev_macros.h`. This macro interface can be used to handle all of the assignments associated with a `tiga.device` request, but the application must explicitly retrieve the return value if there is one from `tiga.io_Return`. Thus, an alternative form of the above program would include the line:

```
#include (tiga/dev_macros.h)
```

and replace lines 20-23 with the following line:

```
set_videomode(TIGA, INIT — CLR_SCREEN);
```

Since the TIGA function `set_videomode` has a return value, if the application program wishes to examine this return value, it must retrieve it from `tiga.io_Return`, as in the non-macro version of the same example program.

Notice that the macro definitions all use the synchronous `DoIO` interface function, rather than an asynchronous function. It is possible to use an asynchronous interface if the application ensures that an `IORequest` is not reused until the `tiga.device` has completed the previous device command requested.

For more information on programming the A2410, refer to the *A2410 Programmers Reference Manual*, and the *TI34010 Users Guide* and *TIGA Reference Manual* distributed by Texas Instruments.

—

—

—



A3000 System Architecture

The Amiga 3000 represents the next logical step in the progression of hardware platforms for the Amiga computer family. The A3000 integrates as standard equipment all of the advanced features which previously had to be added to the the A3000's predecessor - the Amiga 2000. By incorporating all of these features into the base platform via five new custom gate arrays, system performance has been maximized and the cost to the consumer minimized. The result is a compact, yet extremely powerful new platform which represents the new Amiga standard from which newer and even more advanced peripherals will proliferate.

The following is a list of some of the key new features of the A3000:

- 68030 CPU (16 or 25 Mhz).
- 68881/2 Math Coprocessor.
- 1 Megabyte of 32 bit CHIP RAM standard (internally expandable to 2 megs).
- 1 Megabyte of 32 bit FAST RAM standard (internally upgradable to 16 megs).
- 1/2 meg of 32 bit ROM.
- De-interlaced RGB video.
- 32 bit SCSI DMA hard disk controller (with 40 Mbyte disk).
- 4 ZORRO III expansions slots.

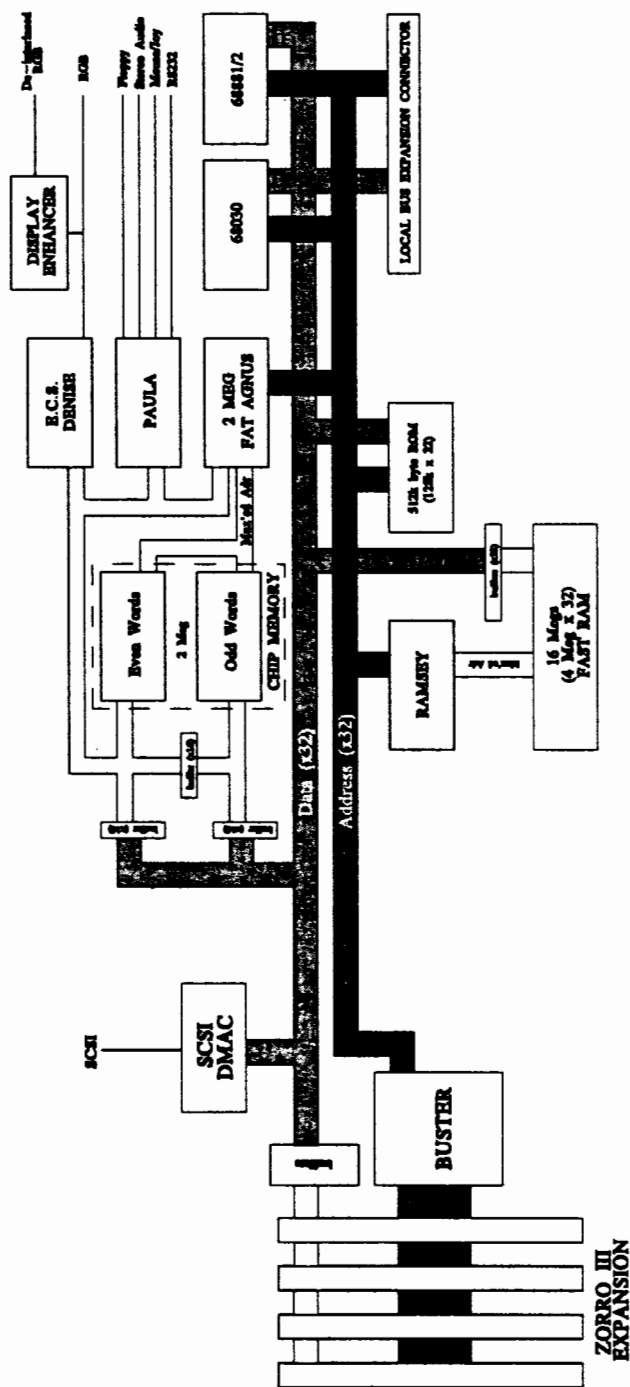
A block diagram of the A3000 is shown on the next page.

A memory map of the A3000 is shown on the second page following.

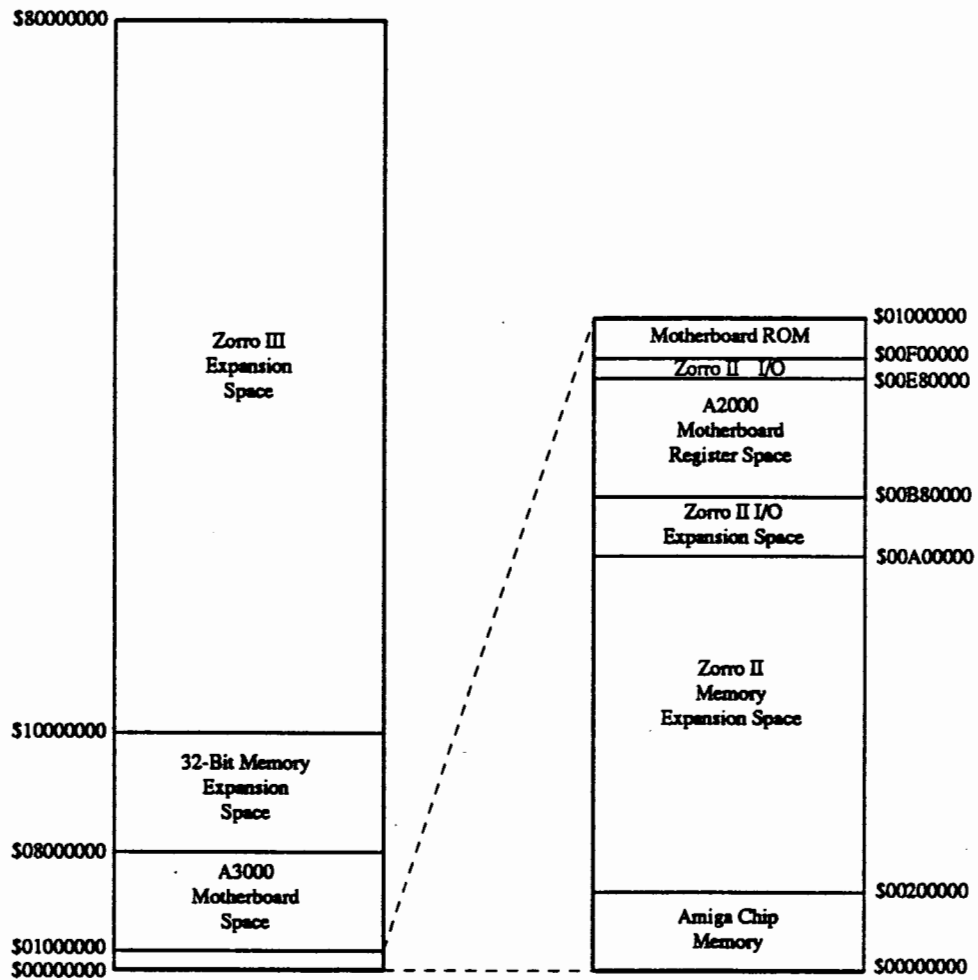
SYSTEM CONFIGURATIONS

At the time of this writing, three different configurations of the A3000 are available. They differ in math coprocessors, system clock speed and hard disk capacity.

The first configuration has a system clock speed of 16 Mhz, a 68881 math coprocessor and a 40 megabyte hard disk. The second has a system clock speed of 25 Mhz, a 68882 math chip



A3000 BLOCK DIAGRAM



A3000 MEMORY MAP

and a 40 megabyte hard disk. The third configuration is the same as the second, except for a larger disk capacity (105 megabytes).

Although the only difference between a 16 Mhz system and a 25 Mhz system is the 68030, 68881 and a crystal, it would be extremely difficult to upgrade the speed of the former. The 68030 and the 68881 are both soldered directly to the motherboard. Suffice it to say that there are very good reasons why this was done, so they may not be obvious. Upgrading a 16 Mhz machine to a 25 Mhz version can most easily be done by adding a daughter card in the local bus expansion slot which need only contain a 68030, 68881/2 and an oscillator.

FAST MEMORY & RAMSEY

Sockets are provided on the motherboard of the A3000 so that the user may easily increase the memory in his machine up to a total of 16 megabytes of FAST 32 bit memory. The motherboard is socketed to accept up to 32 DRAM IC's in a ZIP package style. There are also 8 DIP sockets which can be used. The DIP sockets are electrically equivalent to the first bank of ZIP DRAMs. Consequently, the DIP sockets and first 8 ZIP sockets cannot be populated concurrently. The A3000 is shipped with 1 megabyte of FAST memory installed. The RAM chips used are 80 nanosec (or less) 256k by 4 IC's, and are installed in the DIP sockets.

The DRAM controller (RAMSEY) is designed to work with either 256k x 4 DRAMs or 1M x 4 DRAMs. ZIP sockets were used on the motherboard because 1M x 4 DRAMs are not currently available in .3 inch DIP packaging. The DIP sockets were included so that Commodore could use DIP DRAMs when producing the machine, since they are more readily available in quantity (they're a tad cheaper currently as well...). If all 32 locations are filled with 256k x 4 parts, then the total FAST RAM is 4 megabytes. Using 1M x 4 parts, the total would be 16 megabytes. 1M x 4 and 256k x 4 parts cannot be intermixed on the same motherboard in the FAST RAM!

The use of static column mode DRAMs allows for increased system performance (allows 'burst mode' and 'static column mode' to be used). This type of memory is therefore recommended. ALL of the RAM must be the static column type for this to take affect (the operating system checks at bootup if all of the RAM is the static column type).

RAMSEY Modes of Operation :

The FAST memory subsystem of the A3000 is controlled by the RAMSEY gate array. The Control logic in RAMSEY was designed to support different modes of operation so that

maximum system performance could be achieved. In addition to standard synchronous 68030 bus cycle RAM accesses, RAMSEY supports 68030 burst read mode. RAMSEY also has a special mode of operation referred to as 'static column' mode. Static column mode is designed to reduce the average number of clock cycles required when the 68030 does synchronous bus cycle accesses to the FAST RAM. Burst and Static Column Modes require that the FAST memory uses static column mode DRAMs only!

Standard Synchronous Bus Cycles:

In this type of operation both Static Column and Burst modes are disabled. This mode requires page mode type DRAMs only. Access to the FAST RAM always takes 5 clocks at 25 Mhz, and 4 clocks at 16 Mhz.

Static Column Mode:

This mode requires static column mode DRAMs. When RAM is first accessed, RAS is held low after the cycle completes. This leaves the current RAM page 'open', and the chip will act like a static RAM on this page. Any data in this page can be accessed by simply changing the column addresses only. Since the column address access time (tAA) is much less than the RAS access time (tRAC), subsequent accesses to this page of data can be done faster. As long as RAS is held low (10 usecs max) RAMSEY will allow the CPU to access RAM on this page in only 3 clocks (16 and 25 Mhz).

Comparators inside RAMSEY monitor the ROW address of RAM accesses. If a page is currently 'open', and the ROW address matches (page hit), the RAM can be read in 3 clocks. If the comparators detect that the data being requested is on a different page (page miss), then RAS must be cycled high and low again (tRP), opening up a new page of RAM. Since RAS must be cycled when a page miss occurs, RAM accesses take longer (7 clocks at 25 Mhz, 5 clocks at 16).

There is some difference in how page mode is done at 16 Mhz versus 25 Mhz. At 25 Mhz the page comparator only detects page misses when *AS (68030 address strobe signal) is low, and the RAM is being addressed. Therefore, when a page is opened (RAS held low), it will remain open until the next refresh occurs, or a page miss is detected. At 16 Mhz, however, the page comparator will detect a page miss while *AS is high. Therefore, at 16 Mhz a page will stay open as long as consecutive bus cycles access RAM in this page (or a refresh occurs). At 25 Mhz the page will remain open even if bus cycles in between accesses to the currently opened page occur (such as CHIP memory, CIA's, etc.). This was done so that page misses at 16 Mhz will only take 5 cycles - addresses are valid one-half cycle before *AS goes low (30 nanosecs). If it waited until *AS was valid before detecting a page miss, the RAS precharge requirement (tRP) could not be met in 5 clocks.

Burst Mode:

In this mode, RAMSEY will respond to the *CBREQ (cache burst request) input from the 68030 and allow burst accesses to FAST RAM. Each burst cycle takes 2 clocks each (up to three).

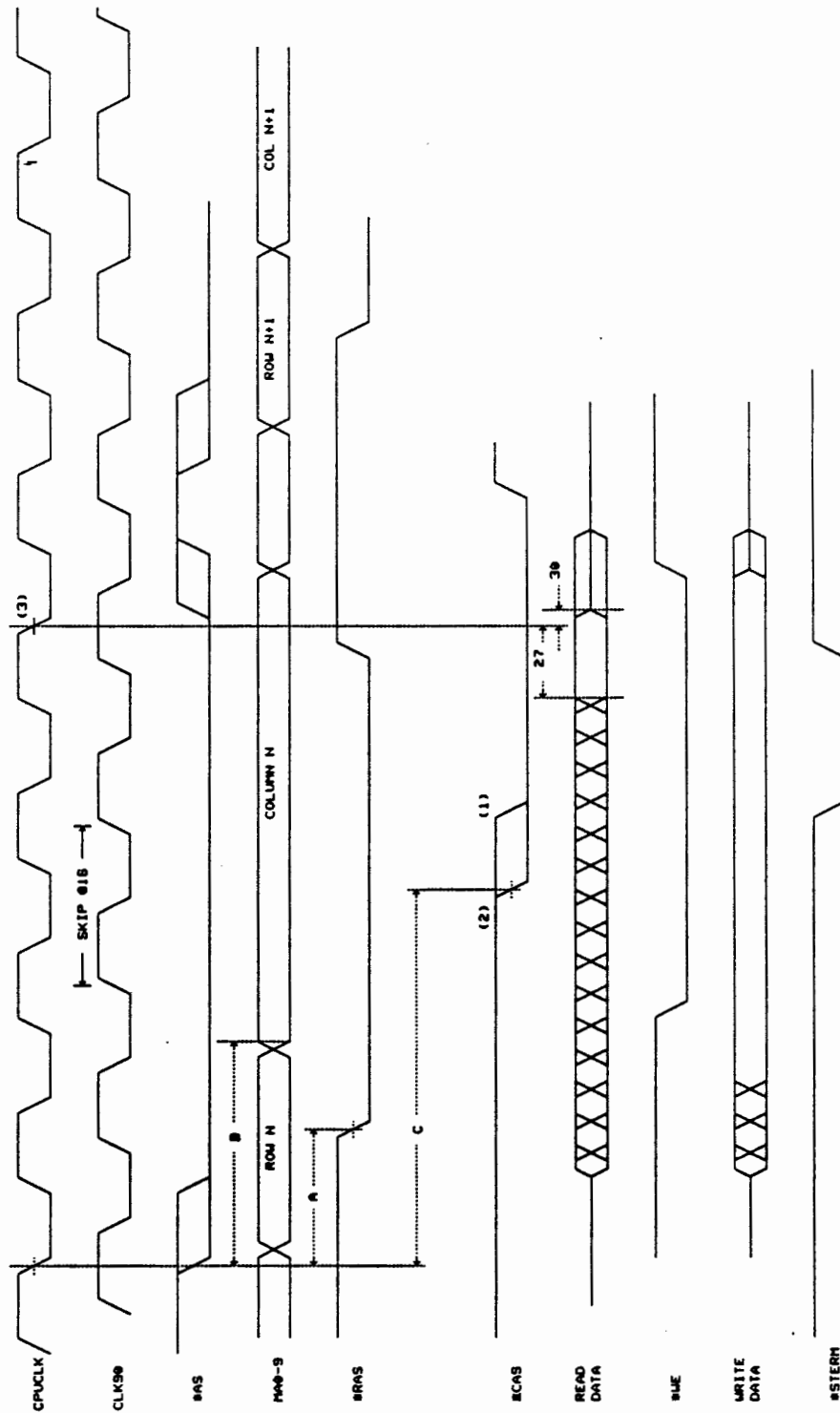
RAMSEY allows for additional control over burst cycles via the WRAP option. The 68030 will always request 4 longword values during a burst sequence. However, if the initial longword is not aligned on a quad longword boundary (A3,A2 not equal to 0,0), the 68030 will read in data which is behind the first data it asked for. Since it is less likely that this data will actually be needed, RAMSEY can stop the burst sequence after the data with A3,A2=1,1 is accessed.

Burst mode requires the use of static column mode DRAMs. Diagrams 4 and 5 show timing for burst accesses of FAST RAM.

RAMSEY Control Register:

There is a single 8 bit register internal to RAMSEY which can be used to change its mode of operation. This register is readable and writable. It is located at \$00DE0003 of the supervisor data space. Data written into the register does not take effect until the next refresh occurs. Consequently, if you write a value to the register you will have to wait out the refresh interval before it can be read back. Each of these bits has a default value that it is set to when the system is reset (actually, a software induced reset will not currently affect these bits in the A3000).

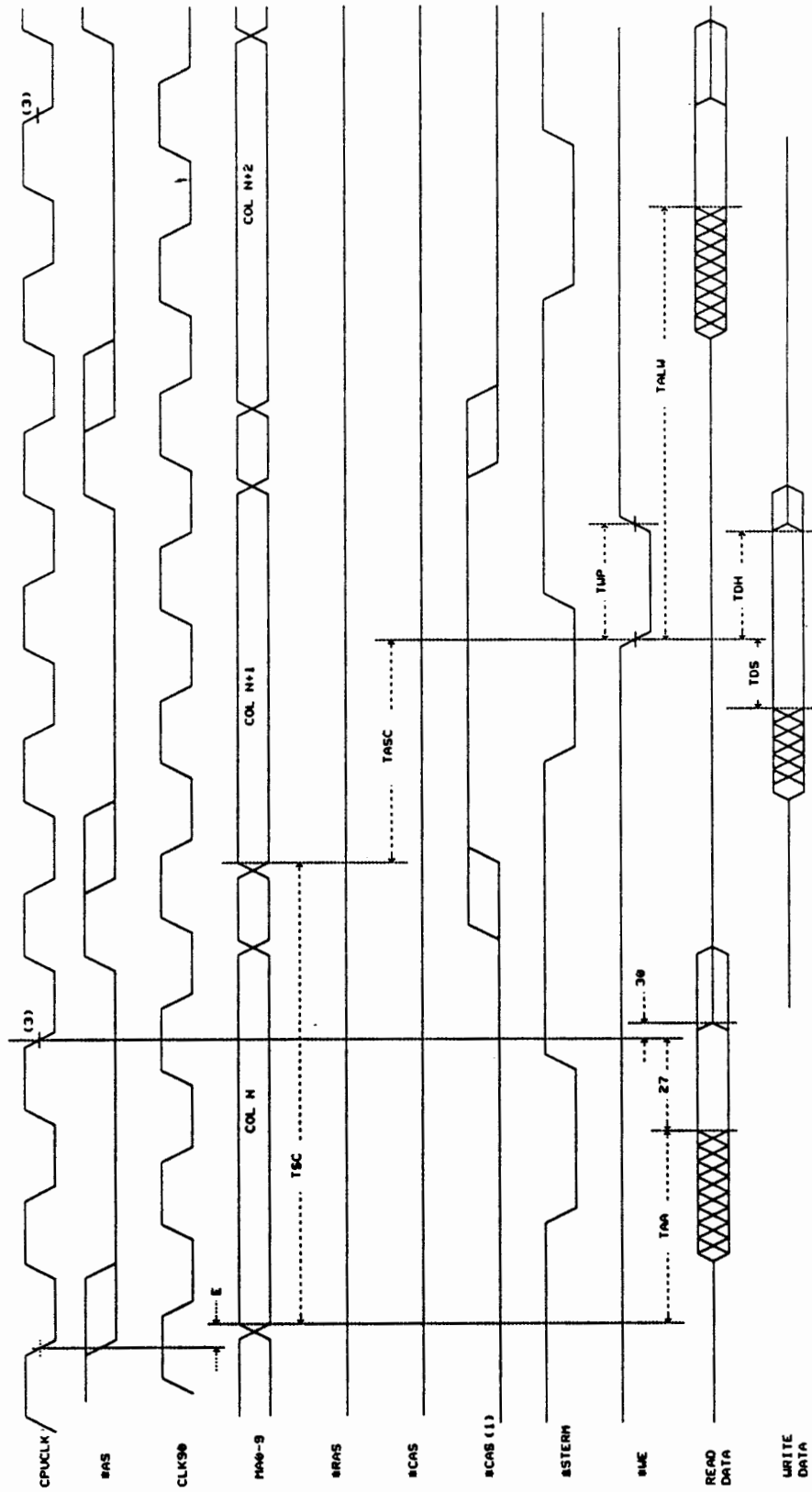
bit 0	ST. COL. MODE	When high, Static Column mode is enabled (def.=0).
bit 1	BURST MODE	When high, RAMSEY will respond to the *CBREQ input and do burst cycles (def.=0).
bit 2	WRAP	If high, all 4 longwords of a burst will be allowed to take place. If WRAP is low, then the burst will only continue while A3,A2 are increasing - burst will not be allowed to WRAP to A3,A2=00 (def.=0).
bit 3	RAMSIZE	If low, then RAM is 1 megabit (256k x 4 or 1M x 1). If high, then RAM is 4 megabit (1M x 4). The default size is determined by the input signal on the RSIZE pin (connected to J852 on the A3000). (1M x 1 not supported in the A3000)
bit 4	RAMWIDTH	If low, RAM is 1 bit wide (1M x 1). If high RAM is 4 bits wide (256k x 4 or 1M x 4). (def.=1)
bit 5,6	REFRESH RATE	The refresh counter uses the CPUCLK to count out refresh times. The number of clocks



NOTES

- 1 - 15 MHz
- 2 - 25 MHz
- 3 - CPU READS DATA

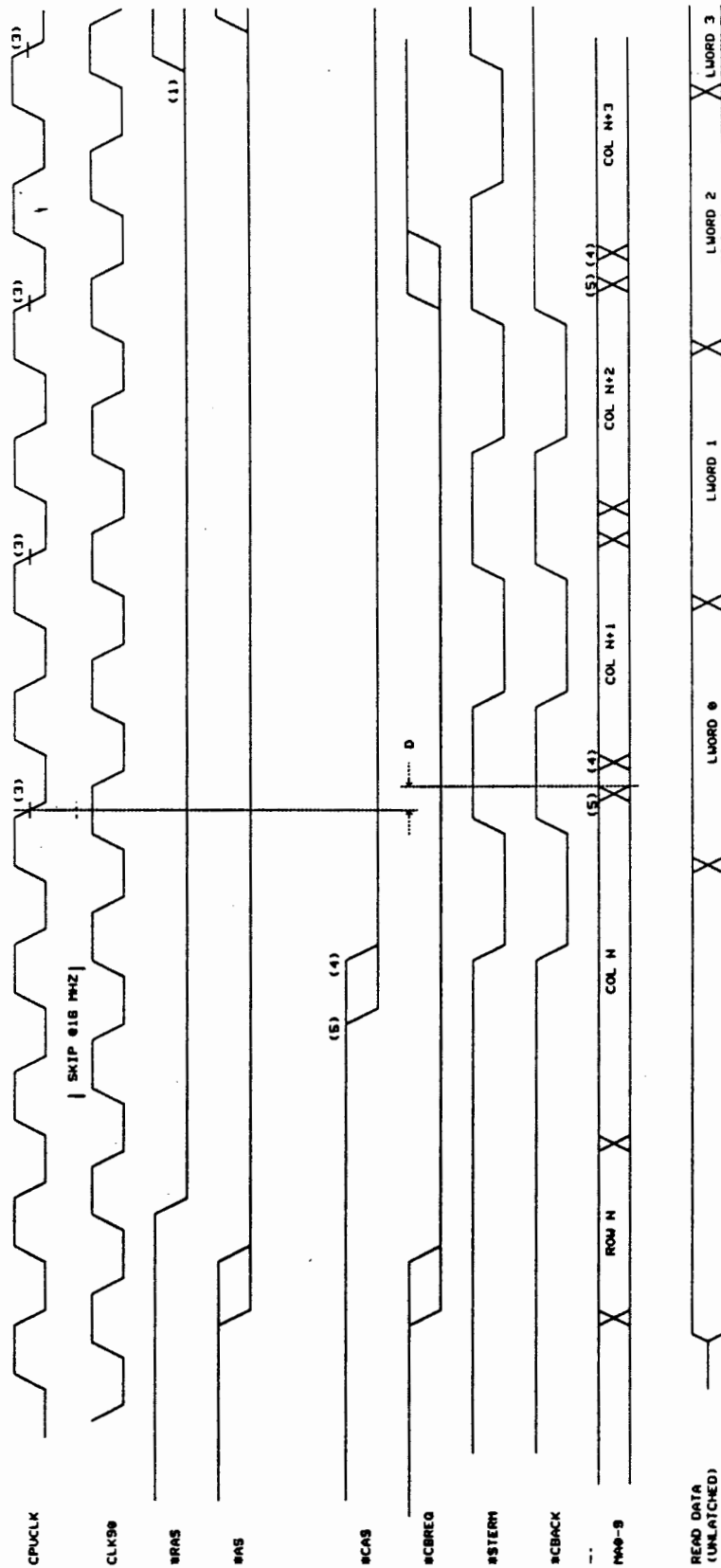
STANDARD RAM ACCESS (Page Mode Off)



* NOTES

- 1 - THIS #CAS SHOWS TIMING FOR BYTES NOT TO BE WRITTEN
- 3 - CPU READS DATA
- 4 - 16 MHz
- 5 - 25 MHz

RAM ACCESS (PAGE HIT, MIXED READ/WRITE)



NOTES

- 1 - GOES HIGH ONLY IF PAGE MODE IS DISABLED
- 3 - CPU READS DATA
- 4 - 16 MHz
- 5 - 25 MHz

BURST READ - NO PAGE OPEN

between refreshes is determined by the following table:

bit 6,5	# of clocks	interval (usecs)	
		16 Mhz	25 Mhz
00	154	9.24	6.16
01	238	14.28	9.52
10	380	22.8	15.2
11	00	(refresh off)	

Since 512 refreshes must be done in 8 msecs, the interval between refreshes must be less than 15.625 usecs. During st. col. mode, RAS can only be low for 10 usecs at a time, so the refresh rate should be set to less than 10 usecs when st. col. mode is enabled. The default values are determined by the CPU SPEED jumper (J851). If low, def. for 6,5=0,0. If high 6,5=0,1.

bit 7 TEST

Leave this bit alone!

Detecting Static Column DRAMs:

When the A3000 is first booted, RAMSEY has burst and static column modes disabled. Burst or static column mode must NOT be enabled unless all of the FAST RAM is the static column type. At boot up, the Operating System examines RAM and determines if it has static column DRAMs only. Each bank of RAM (1 meg with 256k x 4, or 4 meg with 1M x 4) must be checked individually. The proper method for checking for static column DRAMs is the following:

- 1) Disable all interrupts.
- 2) Turn page mode on by setting the bit in the RAMSEY control register (read it back until the bit takes affect).
- 3) Write \$5AC35AC3, \$AC35AC35, \$C35AC35A, \$35AC35AC to 4 consecutive longwords in the same page (A11-A31 must be the same for all 4 longwords).
- 4) Turn page mode off by resetting the bit in RAMSEY (read it back until it takes affect).
- 5) Compare the 4 longwords values with what they were written with. If they are correct, then this bank of RAM has all static column DRAMs.
- 6) Repeat steps 2 thru 5 for each bank of FAST memory.
- 7) Re-enable interrupts.

The code that executes this test must not be in FAST RAM. Any access to FAST RAM with static column mode turned on could cause corruption if any of the RAMs are not static column type. Also, since a refresh cycle will close any open page, writes to the 4 longwords

of RAM must be less than 10 usecs apart.

Since the operating system checks this for you, there should be no reason to do this yourself. If you want to turn on a special mode it should be done through the O.S.

RAM Wait States:

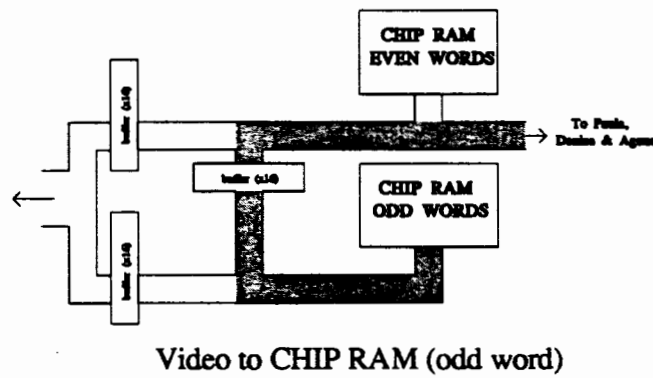
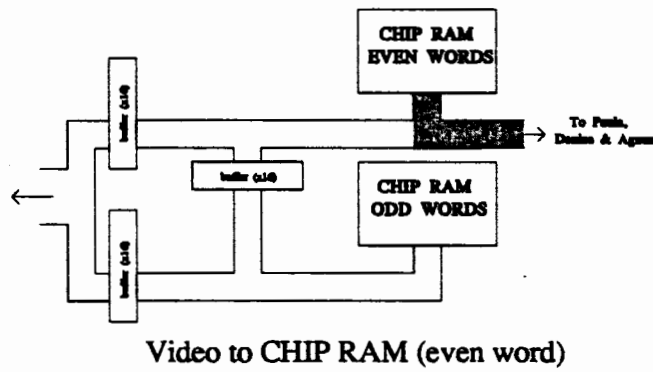
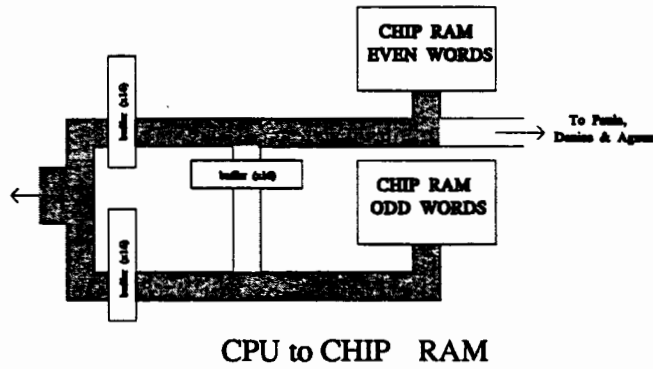
When discussing the performance of a particular computer, sooner or later the question "How many wait states does it have?" will come up. There is no straightforward answer to this question since there are so many factors involved. Furthermore, wait state comparison is only valid when comparing machines using the same CPU running at the same speed.

In order to understand where the wait states come from in the A3000, one must first understand the operation of the 68030's bus interface. The 68030 has the ability to access external memory in three basic ways - synchronous, asynchronous and burst. Synchronous bus cycles talk to 32 bit wide ports only, and have the potential to be completed in as little as 2 clocks. Asynchronous bus cycles talk to 8, 16 or 32 bit ports, and have the potential to complete in 3 clocks. Burst accesses consist of up to four synchronous bus cycles, the first of which can take as little as 2 clocks, and the remaining three in 1 clock each.

Standard synchronous bus cycles to FAST RAM in an A3000 take 5 clocks on a 25 Mhz machine, and 4 clocks on a 1 Mhz machine. Since synchronous bus cycles have the capability to complete in two clocks, the 25 Mhz machine has 3 wait states, and the 16 Mhz machine 2 wait states. Even though the 16 Mhz machine only has 2 wait states, it is still slower than the 25 Mhz machine with 3 wait states. 5 clocks at 25 Mhz equals 200 nanosecs, whereas 4 clocks at 16 Mhz is 240 nanosecs. This illustrates why wait state comparisons are only meaningful on machines operating at the same speed.

When Ramsey is in Static Column Mode, a page hit results in a bus cycle taking 3 clocks (for both 25 and 16 Mhz). This results in 1 wait state. Page misses take 7 clocks (25 Mhz) and 5 clocks (16 Mhz), for a total of 5 and 3 wait states. If a page was not already open, then the rules governing a standard synchronous cycle apply.

The first bus cycle of a burst sequence is a standard synchronous bus cycle, so the number of wait states depends on whether or not static column mode is enabled, and whether a page is hit, missed or not opened. The subsequent cycles (up to 3) take 2 cycles each, resulting in 1 wait state each.



CHIP MEMORY

The A3000 can access up to 2 megabytes of chip RAM. Although all current 8372 FAT AGNUS chips used in the A500 and A2000 are designed internally to support 2 megabytes of CHIP memory, there are separate 1 meg and 2 meg versions. These two versions differ in pinout only. The 2 meg version is designed to be used with 1 meg x 1 DRAMs, and therefore requires an additional multiplexed address bit out (DRA9) and a single *RAS out. Also, in order to access the second meg of memory, another address bit from the CPU (A20) must be fed into the chip. The following table shows which pins have changed.

pin	1 meg	2 meg
35	XCLKIN	A20
56	*RAS1	DRA9
57	*RAS0	*RAS

Since the the external clock input pin (XCLKIN) was sacrificed, the clock switch mechanism must be done externally (it's done in FAT GARY).

With a single *RAS and *CAS (*CASL, *CASH), using 1 meg x 1 DRAMs allows for 2 megs of CHIP memory configured as a single bank of 1 meg x 16. This configuration is not desirable for two reasons. First of all, since the 2 megs are contained in a single bank, the machine would have to ship with 2 megs of memory since this is the minimum (as well as maximum) configuration. Obviously we all would like to have 2 megs of CHIP RAM, but this would significantly affect the selling price. The second problem with 1 meg x 1 DRAMs is the width of the 68030 interface to it. Although the custom chips can only access CHIP RAM 16 bits at a time, the 68030 wants to talk to CHIP memory 32 bits at a time. If we limit the CPU interface to 16 bits, throughput will be significantly reduced. So now we need CHIP memory that is 32 bits wide. Using 1 meg x 1 DRAMs, we would need 32 chips, which results in 4 megabytes of CHIP RAM. Since the custom chips can only access 2 megs, this obviously won't work.

256k x 4 DRAMs were used to solve these two problems. Eight 256k x 4 DRAMs yields 1 megabyte of memory that is 32 bits wide to the 68030, and 16 bits wide to the custom chips. An additional bank of 8 chips can then be added to double the amount of CHIP RAM.

Since AGNUS only supplies two *CAS signals (*CASH, *CASL), external logic is needed to generate the 8 separate CAS'es that are needed. In order to do this, however, we need to know which bank is being asked for. It's easy to tell what the CPU is asking for since A19 and A20 are available, but the addresses that the custom chips want is internal to the AGNUS chip itself. The proper bank must be determined by 'extracting' A19 and A20 from the

multiplexed address coming out of AGNUS. A19 is extracted by latching the value of DRA9 at *RAS time. DRA9 contains the value of A20 at *CAS time.

Figure 6 shows the different possible data paths for CHIP RAM data.

SYSTEM ROM

The A3000 has 512k bytes of ROM configured as 128k x 32. Rom timing is adjustable via J152 and J151. The proper timing for best performance is set when the machine is manufactured and should not be changed unless ROMs of different speed are used.

J152	J151	Clocks
0	0	5
0	1	6
1	0	7
1	1	8

The minimum output enable (Toe) and access (Tacc) times for the ROMs is determined by the following:

$$\begin{aligned} \text{Toe} &= ((\# \text{ cpu cycles}) - 2) * \text{Tcyc} - \text{Tgary} \\ \text{Tacc} &= ((\# \text{ cpu cycles}) - 1) * \text{Tcyc} \end{aligned}$$

where

$$\begin{aligned} \text{Tcyc} &= \text{period of a single 68030 clock} \\ \text{Tgary} &= \text{max delay thru GARY (30 nsecs)} \end{aligned}$$

DISPLAY ENHANCER

The A3000 has an additional video output port which contains 'enhanced' video information. The Display Enhancer circuitry improves the quality of the video display by removing flicker and visible scan lines from interlaced, noninterlaced and ECS graphics modes. The display enhancer is 100% transparent to system software and is compatible with all video output modes (in superhires mode the bypass should be enabled via the switch).

32 BIT SCSI DMA

A SCSI is built into the A3000 as standard equipment. The custom DMA controller (DMAC) connects the 8-bit SCSI world to the 32 bit bus of the A3000. Data is buffered

between the two buses with a FIFO internal to the DMAC IC. The DMAC requests the A3000 system bus only when its FIFO is full, and transfers the 32 bit data at full system bus bandwidth. This method minimizes the amount of time that the DMAC occupies the bus, and allows other concurrent operations to continue at their maximum efficiency.

Because of pin limitations, the DMA address counters are contained in RAMSEY.

ZORRO III EXPANSION

Four ZORRO III expansion are provided in the A3000 for add in cards (there are 2 IBM compatible slots as well). The ZORRO III bus definition is a superset of the ZORRO II bus of the A2000. It is designed to be compatible with existing ZORRO II peripheral cards, as well as new cards which can be designed to take advantage of higher speed, address ranges and data widths. Translation of 68030 local bus signals into waveforms compatible with ZORRO II cards (and vice versa) is the responsibility of the BUSTER gate array.

VIDEO EXPANSION

A video expansion slot that is compatible with that of the A2000 is provided in the A3000. The form factor for this card is slightly different since it is in-line with one of the ZORRO III expansion slots. An adapter plate is provided to allow A2000 size video expansion cards to be used in the A3000.

BUS MONITORING

The FAT GARY gate array contains logic that monitors the local bus for a lengthy bus cycle. If a bus cycle lasts longer than a predefined interval, then FAT GARY will terminate the cycle. This helps to inform the system of accesses to areas of memory that are undefined, and discourages the design of peripherals that are inefficiently designed to operate in a multitasking environment. The A2000 had automatic bus termination built in as well, but its purpose was vastly different. The A2000 would cause a normal termination of a no-wait state bus cycle, unless it was told to wait. The A3000 provides no such default bus cycle termination. In general, if the FAT GARY has to be called on to terminate the bus cycle, then something is wrong!

After the assertion of *AS, a counter in GARY starts running, and is reset by the de-assertion of *AS. If the counter counts down before *AS is de-asserted, GARY steps in to terminate the cycle. There are two different timer values available, each of which terminates the cycle

differently.

There is an 8 bit register in GARY at \$00DE0000 of the supervisor data space. When written to, bit 7 selects the timeout mode to be used. Writing a 0 to this bit enables DSACK timeout, and a 1 enables BERR timeout (after a RESET, DSACK timeout is enabled). DSACK timeout counts for 32 C! pulses (approximately 9 usecs), and then asserts both DSACKs to terminate the cycle. BERR timeout takes much longer, counting for approximately 250 msecs before asserting the *BERR signal. Whenever a bus timeout occurs in either mode, bit 0 of the register at \$00DE0000 is set, and is not reset until the register is read.

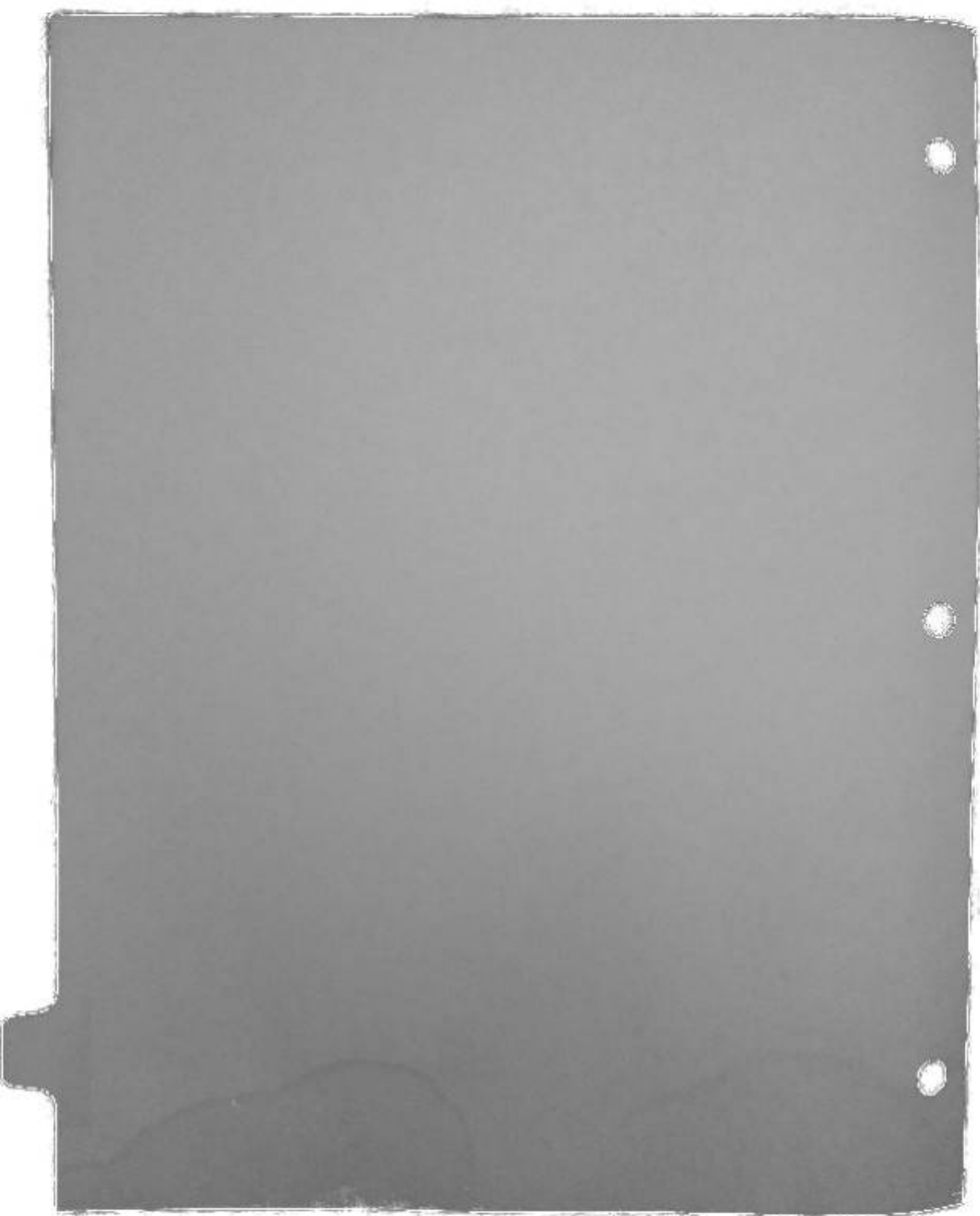
The DSACK timeout mode was made much shorter, and terminates the cycle more discreetly using DSACKs. This mode was made the default because of compatability issues. The 1.3 ROMs purposely snoop through a large range of addresses during boot-up which are normally not there. Taking 250 msecs for each one would take forever (not to mention the *BERR causing GURUs...). The idea is to get everything up and running, and then change over to BERR mode.

Since the blitter has the capability of keeping the CPU off of the CHIP bus for a long time, bus timeout detection is disabled whenever CHIP RAM or CHIP registers are being selected.

Automatic bus timeout can be disabled altogether by writing a 1 to bit 0 of the 8 bit register at \$00DE0001. After a RESET, this bit is automatically set to 0 (timeout enabled).

POWER UP (COLD START) DETECTION

There is a bit in GARY which detects whenever the power supply dips below 4.5 volts. Hopefully, this should only happen as the machine is powering up. Bit 0 of the 8 bit register at \$00DE0002 will go high whenever this happens. The bit can only be reset by writing a 0 back to it.



Font Independent User Interfaces

by Micheal Sinz and James Barkley

For software to be font independent, it must not rely on a specific font style or size in its user interface presentation. This lets the user select the font for the display at run-time, either due to having set the system's default font via preferences or as an option in the program.

With AmigaOS 2.0, one of the Amiga's basic features, its user customizable environment (via the Preferences program) has become even more user configurable. One of the customizations now possible is the ability to set the default screen and system fonts. With the addition of these features, the burden is now on the application developer to correctly operate with the user's preferences.

User selectability of font size was added not only to lengthen to the list of customizations available to the user; it is also there to help support the various new screen resolutions available with new software and hardware. It's not likely most people will be able to read an 8-pixel high font on a screen that has 1008 x 800 resolution on a 14-inch monitor, like the A2024 monitor. Also, With display modes that have very narrow or short pixels, the use of a 8x8 font would produce strange and sometimes unreadable results.

The purpose of this paper is to give insight into why and how to develop font independent user interfaces. It is understood that building interfaces that are font independent can be more work than building font dependent interfaces. The intent here is to show some of the technical aspects of font independence; but more importantly, to help bring to light some of the issues that need to be thought of while designing the interface to help facilitate font independence. The work that is put into an application to make it font independent will pay off as display resolutions change and users start to demand that the applications use the font of their choosing.

First there should be some technical details as to what is going on in the system as far as fonts are concerned. There are various ways used in different parts of the Amiga to specify the font that text is to be rendered in. The naming convention can also be rather confusing.

Several data structures must be understood in the subsequent discussion. They are:

TextFont

A structure that identifies a font in memory and that is ready for use. A pointer to this structure is return by **OpenFont()** and **OpenDiskFont()**.

TextAttr

A short description that specifies a requested font. This structure is what is used to **OpenFont()** and **OpenDiskFont()** a **TextFont**. This structure is most frequently used by Intuition in the context of **IntuiText** structures and thus may cause some confusion.

IntuiText

A "package" (structure) of information which Intuition uses for most of its text rendering. This structure contains a **TextAttr** font specification in a field named **ITextFont**. Note that this is not a **TextFont** but a **TextAttr**. If the **TextAttr** is NULL or unopenable, some default will be used.

RastPort

A "Graphics Handle" that is to rendering on part of display much like a "File Handle" is to I/O on a file. The **RastPort** must have a valid **TextFont** structure associated with it.

BitMap

The lowest level graphics structure. (The "Raw" graphics data) This is much like the disk drive is, when you take the analogy of **RastPorts** being like file handles. They have no idea what a font is.

To understand what happens when a character string is rendered, the text and font specific calls should be understood. These are documented in the *Amiga ROM Kernel Manual: Libraries and Devices*. Also, a short, basic description of each can be found in the *Amiga ROM Kernel Manual: Includes and AutoDocs*. The new 2.0 font calls are in the AutoDocs on the AmigaOS 2.0 developer disks. Below are quick descriptions of a few of the calls that will be needed in this talk.

void Move(struct RastPort *rp,SHORT x,y)

This moves the current pen position for the **RastPort** to the coordinates [x,y] in that **RastPort**.

void SetFont(struct RastPort *rp,struct TextFont *font)

This function sets the font for a **RastPort**. It updates all of the information in the **RastPort** structure so that text rendering will be possible. A valid **TextFont** must be used.

void Text(struct RastPort *rp,UBYTE *str,SHORT length)

This is the base text rendering routine. It renders the first **length** characters at ***str** into the **RastPort** **rp**. The text is rendered with the **RastPort**'s **TextFont** with the base line of the text at the current **RastPort** pen position. Note that the base line of a font is defined in the font itself and should be the line below which the descenders are. It is *not* the upper most or lower most part of the font.

void PrintIText(struct RastPort *rp,struct IntuiText *itext,SHORT x,y)

This is the Intuition text rendering call. It uses the **IntuiText** structure to define the text to be rendered. The **[x,y]** position is the offset within the **RastPort** that the **IntuiText** structure will be rendered at. The **RastPort**'s pen position does not affect this call. Note that the **IntuiText** structure has another offset that is added to the **[x,y]** position of the **PrintIText** call. Note also that the text position is based on the top-most part of the character and not the baseline like the **Text()** call is. This call eventually calls **Text()** after it has done all work needed.

struct TextFont *OpenFont(struct TextAttr *)

This call returns a **TextFont** that was specified by the **TextAttr**. This call only will return **TextFont** structures that are currently in memory. (Either ROM fonts or currently loaded Disk fonts).

struct TextFont *OpenDiskFont(struct TextAttr *)

This call returns a **TextFont** that was specified by the **TextAttr**. This call will also search for the "best-fit" font on disk and load it and then call **OpenFont()** on it. You *must* be a process to call this function. (In fact, since this is in *diskfont.library*, a disk based library, you must be a process to even have access to it).

void CloseFont(struct *TextFont)

This function closes a previously opened font. It closes fonts that were opened via **OpenFont()** and **OpenDiskFont()**. You should take care to correctly match your opening and closing of fonts. Closing a font too many times can cause a crash while not closing it enough times could cause a waste of RAM (and other problems).

void InitRastPort(struct RastPort *rp)

This call initializes a **RastPort** structure, clearing all the fields to zero or their defaults. In particular, it sets the font to the system's default font: **GfxBase->DefaultFont**.

Note that when displaying text via an **IntuiText** structure or when a **TextAttr** is used to specify the font to be used, the system will only call **OpenFont()** and not **OpenDiskFont()** (This applies mainly to Intuition).

When a window is opened, the window's **RastPort** is initialized via **InitRastPort** and this gets the **GfxBase->DefaultFont**. Thus, any rendering into the **RastPort** via the **Text()** call will render in this font unless you do a **SetFont()**.

Intuition does its rendering with the screen's font, which can be found in three different locations:

- ☐ **Screen->Font**
- ☐ **Screen->RastPort.Font**
- ☐ **Screen->BarLayer->rp->Font**

These should describe the same font.

When a screen is opened, these are initialized by the **TextAttr** in the **NewScreen** structure. If this is **NULL** or can not be opened via **OpenFont()**, the font used will be **GfxBase->DefaultFont**.

Jim Mackraz's paper on fonts from the 1988 DevCon contained a list of text rendering done by Intuition using the triumvirate of screen font pointers listed above. Here is an updated version of that list (with some notes):

<u>Situation:</u>	<u>Notes:</u>
Gadget text	1
Gadget highlight text	1
String gadgets	2
Menu titles	3
Menu items and sub-items	1
Menu equivalent keycodes	4
Menu size calculations	1
Window titles	3
Screen titles	3
Requester text	1
Requester gadgets	1
Alerts	5

Notes:

1. Since rendering happens with **IntuiText**, the **TextAttr** in the **IntuiText** structure will be used if it is not **NULL** and Intuition was able to open that font via **OpenFont()**
2. The text in the string gadget display where the editing takes place is not controlled via **IntuiText**. The normal gadget text is.
3. Not **IntuiText** controlled.
4. Menu shortcuts (command keys) are displayed in the font of the menu item they are connected to.
5. Alerts are forced to be displayed in **Topaz-8**. (**TOPAZ80**)

As the above list implies, any time that your interface needs to have a certain font, it should specify it when possible. Thus, **IntuiText** structures should always be filled in with the font you wish to use. This is much safer than to rely on the current intrinsic operation of Intuition.

The screen font plays a big part in the Intuition text rendering. On the Workbench 2.0 screen, the user can select the font that is to be used as the screen font. For this reason, the Workbench screen's font should be considered the font of choice by the user. This font can be almost any font the user wishes to set it to. Some applications require a mono-spaced font. For this reason, there is a special user selected font that is defined to be mono-spaced. This font is **GfxBase->DefaultFont**. (Before 2.0, this font was only changed via some "hacks" that stuffed values directly into it). Note that this is the font that **RastPorts** are initialized to by default. There is no correspondence between **GfxBase->DefaultFont** and the Screen font descriptors other than that when a screen is opened without a specific font, it gets **GfxBase->DefaultFont**.

If your application is to correctly adjust itself to a font, you will need to select which "default" font to use. Some general guidelines:

1. If you execute on the Workbench screen and can use proportional fonts, use the Workbench screen's font.
2. If you execute on the Workbench screen and need to have a mono-spaced font, use **GfxBase->DefaultFont**.
3. If you execute on your own screen, use the fonts described above or give the user a method for selecting the font.
4. Have some fall-back font (usually **Topaz-8**) that you can revert to in case the selected font won't fit into your interface.

In order to find out what the default fonts are, you will need to access the correct data structures. On a private screen, this is not a problem, however on public structures, it could be. Under 2.0 you should use **LockPubScreen()** to make sure that the screen is locked in an open state. Since there is no legal way for a program to change the font on a screen that is open without closing it first, this will protect you. Under 1.3, there is no real protection other than **Forbid()/Permit()** while snooping in the screen's structure (However, under 1.3 there are no public screens other than Workbench).

Part of the process of building a font independent user interface is the planning of the interface layout. While font independence is possible with most any layout, proper planning will reduce the difficulty. It helps to observe the relative positions of items on the screen and how they relate to each other. Good visual interface design typically dictates an interface that makes for easy adjustment to different font sizes.

Two example interface-only programs are on the DevCon disks. They show some methods possible to accomplish the task of font independence. These two examples are by no means all encompassing.

An easy method to make the display adjustable is to design it in columns. That is, have all of the display elements (which are referred to here as display areas) laid out in columns and rows. The first example shows an interface of gadgets and text boxes that is broken into three columns of display areas. In this case, one needs only to find the sizes of the various columns (and rows) and then adjust the starting position and the size of each of the display elements accordingly.

The second example shows how the column display area method can be used to divide a single display area into a smaller set of rows and columns. This illustrates that much of the effort required for the layout is spent either figuring out the minimum size of a column based on the sizes need for the display areas or figuring out how to fit the display area into the size given for the column.

The first example was done with pre-allocated gadget structures. This example shows how the structures are then manipulated such that the display is built up correctly for the font of the screen. Most of the work done in the program is just "grunt-work" in that it does not involve tricks or complex formulae.

The second example dynamically allocates all of its structures. Again, the work is just "grunt-work" needed to fill in the structures as they are allocated. The example also shows a way to do a "fall-back" for when the screen's font is too large.

NOTE: In 2.0, much of the gadget creation code is eliminated via the use of GadTools. This considerably reduces the amount of code you need to write. (The example does not use GadTools in order to show more precisely what is going on). In addition to supplying gadgets with a standard look-and-feel, GadTools is also highly recommended because it makes programming easier.

One of the "chores" involved in making a user interface font independent is figuring out the sizes of all of the display areas. A good to deal with this is to break up the interface into rectangles that contain a gadget, box, graphic, or text. The display areas containing default-font text need to be adjusted to the size of the text it contains. Because proportional fonts make it impossible to just blindly look at the string size and multiply by some number, it is necessary to use `TextLength()` or `IntuiTextLength()` to obtain the horizontal size of the text (This is in pixels). Usually, a maximum column width needs to be determined so that all the display areas will be of a uniform column width. This also makes it easy to replace text without changing the interface. If you change a word to a larger word, the code can adjust

for its larger size. If you look at the second example, you will notice that all of the strings for the example are in one source file and they are referenced via a function call. By adding this minor overhead, text substitution for multiple language support has been made rather simple. It could even do this at runtime by reading a file containing the text strings that will be used. (It is rather fun to see what happens in this example when you change some of the text for the gadgets. Note that if the text gets so large that the layout of the window would make it wider than the screen, it will not open).

The work needed to layout a user interface with respect to the user's default font is not very difficult. After the interface has been designed, and the display area relationships established, the rest is just "grunt-work" that take a small amount of time to do. This investment in making your user interface font sensitive will pay off now and in the future as more users utilize the wider variety of display modes and select different default fonts for those modes. In the long run, your software will look much more professional to the users, making them happy and (this is the real reason) produce better sales. After all, wouldn't you prefer to buy the product that worked well *with* the system?



—

—

—



BAP – Basic Amiga Programming

By Kevin-Neil Klop

1 Introduction

The Amiga is one of the most complex microcomputers to program currently in existence. Yet, for all this complexity, it is based on a few simple concepts. Once these basic concepts are mastered, then programming the Amiga becomes immeasurably easier.

This class, these notes, and example code, is designed to familiarize you with these basic concepts. Note that you will not come out of this class as a full fledged Star Programmer, but you will have both feet planted upon the road to that elusive goal – a working Amiga program.

2 Thanks

I'd like to thank Darren Greenwald and Brian Jackson for their help in preparing much of the example code presented in this presentation.

3 Basic Terminology

In order for two people to converse, there has to be a vocabulary that both agree upon; a set of words whose meanings are understood by both parties. In order to insure that both you and I are speaking the same language, and that none of the words used are foreign to you, the following vocabulary list was compiled:

- (1) *Include File* An include file contains text that is to be included within another file. For instance, if the file "FOO.H" contained:

```
I really like the A3000, and will  
buy one as soon as I can.
```

and the file "BAR.C" contained:

```
Unfortunately they cost more  
than I have in the bank.
```

then the preprocessor will convert this to something akin to:

```
I really like the A3000, and will  
buy one as soon as I can.  
Unfortunately they cost more
```


than I have in the bank.

Include files traditionally have an extension of ".i" for assembly language include files, and ".h" for C include files. Thus, if you are working on something that will be assembled, then you would include "exec.i", but if you were working on something that will be compiled with a C compiler, then you would include "exec.h". Modula-2 has its own equivalent of include files called ".DEF" files. .DEF files contain all the information of the C or assembly include files, but in a way that is consistent with the syntax and philosophy of Modula-2.

Most of the time, include files are to define data structures and/or constants that are shared across several modules. For example, rather than declaring the constant, "MYWINDOWSIZE" in each of several dozen modules, it's generally considered better to declare it in ONE file, and then include that file in all the modules that require that constant.

- (2) *defines* A define is a C-language construct that is generally used to define constants or symbolic names for things. For instance, when declaring a screen to be a custom screen (as opposed to a workbench-type of screen), one can either type the number 0x0001, or the word WBENCHSCREEN. The latter is generally considered more informative than the number.

The way a define is generated depends on the language. However, since there are so many examples in C, the syntax for C is shown:

```
#define THECONSTANT THEVALUE
```

- (3) *square brackets* Are used throughout many textbooks and notes to mean, "This is optional." Deviations from this rule will be clearly marked.
- (4) *chip memory* is memory that is shared among the custom chips. It is mainly used for buffers and display memory. Basically, any data that will be referenced by the custom chips must be in chip memory.
- (5) *fast memory* is memory that is accessible only by the 68000 (or 68020, or 68030). As a result, access to it is many times faster than to chip memory since you don't have to share it with other chips.
- (6) *public memory* is a somewhat nebulous type of memory that may either be fast or chip memory. Its basic claim to fame is that it is guaranteed to always be in core at the moment, although there may not be enough for whatever you have in mind.

4 Implicit Assumptions

There are certain assumptions that I made when writing the course materials for this class. The first assumption that I made was that you are interested in learning to program the Amiga computer. If you have these notes, or are attending the accompanying lecture, then this seems a safe assumption.

The second assumption that is being made is that you are familiar with your chosen language, i.e. C, assembly, Modula-2, FORTH, BASIC, Cobol (oops!), or whatever language you are going to use. Trying to learn a language AND the Amiga at the same time will be a pretty tough job. Not impossible, but still pretty tough.

As a result of this second assumption, little or no attempt will be made to teach you a language as we go

unless something non-standard appears in an example. Note that most of the example code available for the Amiga centers around the C language. Indeed, most of the examples in this class will be for C.

Also, the Rom Kernel Manuals are the basic handbook that you need to program the Amiga. The latest ones contain a lot more explanatory information than the previous one. These "RKMs" provide descriptions of all the system calls, data structures, and behaviours that you will likely need in programming the Amiga.

5 Some Basic Types of Things

There are many types of things that are dealt with on the Amiga. From the usual C types of *ints*, *longs*, *chars*, *structs*, and *pointers*, are derived the system wide standard types of *UBYTE*, *ULONG*, *UWORD*, *APTR*, and *BPTR*.

These types are defined in the include file `exec/types.h` or `exec/types.i` depending on whether you are using C or assembly. For Modula-2 people it's in there somewhere, although where it is varies depending on which Modula-2 compiler you use.

The basic meanings of these types are:

- (1) *UBYTE* - an unsigned integer values of one byte (8 bits) in length. In C it is defined as an unsigned char.
- (2) *ULONG* - an undigned long integer of 4 bytes (32 bits) in lengths. In C it is defined as an unsigned long.
- (3) *UWORD* - an unsigned integer 2 bytes (16 bits) in length. In C it is defined as an unsigned int.
- (4) *APTR* - a generic pointer to almost anything except a function. It is generally defined as a "void *" in C.
- (5) *BPTR* - a BCPL pointer. These are a little harder to explain than *APTR*s. They are essentially pointers to thgings, but use a format that is specific to BCPL (the language that DOS was originally implemented in). A *BPTR* is equivalent to an *APTR* shifted right by two bits. Thus, to convert a *BPTR* to an *APTR*, merely *BPTR* << 2. On the other hand, taking a random *APTR* and shifting it right two bits will, three out of four times, result in a bad *BPTR*. This is because objects pointed to by a *BPTR* **MUST** be long word aligned (i.e. the address must be evenly divisible by four) whereas an *APTR* can be any value at all.

With those basic building blocks, as well as the normal C types, almost everything can be understood.

6 Exec And Its Place In The World

Exec is basically the center of the universe for the Amiga operating system. Almost all interactions with the other parts of the Amiga operating system require interaction with some part of Exec. Thus, an understanding of how Exec deals with things gives you an entry into understanding how to deal with the rest of the system.

The following sections will outline for you some of the major areas of Exec that you may need to deal with, and some of the concepts necessary for intelligent use of the Exec functions.



7 Lists

One of the major responsibilities of Exec is the management of many of the system lists. Some of these lists are:

- (1) List of programs that are waiting to use the CPU
- (2) List of memory regions that are currently unallocated
- (3) List of libraries that are currently in memory
- (4) List of devices that are currently in memory

Lists also control many of the other functions that your program will be interested in.

What, then, does a list look like?

A list consists of two parts. The list header, and the nodes on the list. The list header contains information about the list such as the type of things that may be added to the lists and pointers to the first and last nodes on the list. If all that information isn't needed, then there is a smaller structure that consists only of pointers to the first and last nodes on the list. In addition, the list header serves as a kind of "handle" with which to refer to the list as a whole.

A full list header is structured like this:

```
struct List {
    struct Node *lh.Head;
    struct Node *lh.Tail;
    struct Node *lh.TailPred;
    UBYTE lh.Type;
    UBYTE lh.Pad;
}
```

Or, if you don't need all that information, then the structure becomes a minimal list header and is structured like this:

```
struct MinList {
    struct MinNode *mlh.Head;
    struct MinNode *mlh.Tail;
    struct MinNode *mlh.TailPred;
}
```

Nodes can, themselves, be broken down into two parts. One part is common to all nodes, no matter what type they are; whether they are nodes created by the user or whether they are nodes that belong to the system. This part is invariant and used to link nodes together. The second part varies from one node type to the next, and possibly varies even within a single node type.

A full node header is structured like this:

```
struct Node {
    struct Node *ln.Succ;
    struct Node *ln.Pred;
    UBYTE ln.Type;
}
```



```

        BYTE ln_Pri;
        char *ln_Name;
    }

```

and if you don't need all that information, then the structure becomes a minimal node and is structured like:

```

struct MinNode {
    struct MinNode *mln_Succ;
    struct MinNode *mln_Pred;
}

```

As with all things used by a program, the list header and nodes must be initialized before they are meaningful. It would appear to be pretty much straightforward, but the list header and minimal list header have a little bit of a twist in their initialization.

The first two fields of the list header look somewhat like the first two fields of a node – two pointers to Node structures. Also, the second two fields of the List structure resemble the first two fields of a Node structure as well. It turns out that the first two fields of the List structure together form the first node on the list, and the second two fields of the List structure form the last node on the list.

This is accomplished through initializing them in the following manner (assume that MyList is a structure of struct List):

```

MyList.ln_Head = (struct Node *)&MyList.ln_Tail;
MyList.ln_Tail = NULL;
MyList.ln_TailPred = (struct Node *)&MyList.ln_Head;

```

In order to better understand what is going on, let's compare the first two fields of a List structure with the first two fields of a Node structure:

List	Node
struct Node *lh_Head	struct Node *ln_Succ
struct Node *lh_Tail	struct Node *ln_Pred

Figure 1. Comparison of first two List fields with first two Node fields

In this case, the lh_Head acts like the ln_Succ field of a node, and the lh_Tail acts like the ln_Pred field of a node. However, there is no node "before" the head node on a list, so the ln_Pred field of the first node would be set to NULL. Similarly there is no node after the last node on a list. Thus, the ln_Succ field of the last node would be set to NULL as well. If the first two fields of the list header are taken as the first node on the list, then we have to set the lh_Tail field (which corresponds to the ln_Pred field of a node) to NULL. Similarly, if we take the second two fields of the list header as the last node on the list, then we have to set lh_Tail (which corresponds to the ln_Succ field of the last node) to NULL as well.

Then we set the lh_TailPred field (which corresponds to the ln_Pred field of the last node in the list) to point to the previous node in the list, i.e. the head node. Similarly, we set the lh_Head field of the list header (which corresponds to the ln_Succ field of the first node on the list) to point to the next node in the list (which in an "empty" list is the tail node in the list).

1

2

3

This means that a so-called "empty" list still has two fields in it - the list header node and the list tail node, formed by the first couple of fields in the list header itself. Any nodes that might be added to the list are inserted between these two "pseudo-nodes". After another node is inserted into the list, then the lh_Head and lh_TailPred will both point to that node. That node's ln_Pred will point to lh_Head, and the node's ln_Succ will point to lh_Tail.

So, how do you know if you're at the end of the list?

Well, there are two sub-cases of this. The first is the empty list. In this case, lh_Head will point to lh_Tail (i.e. MyList.lh_Head == &MyList.lh_Tail). The other case is the non-empty list in which case the last true node in the list will point to a node whose ln_Succ field will be NULL. Thus, the test:

```
if (Node2BeTested->ln_Succ->ln_Succ == NULL)
{
    /* We are on the last node of the list. */
}
```

Similarly, to test if something is the first node in the list, you would use a test something like:

```
if (Node2BeTested->ln_Pred->ln_Pred == NULL)
{
    /* We are on the first node in the list. */
}
```

8 How to use Lists

What if you're like me and can't remember how to initialize the list headers? Well, you could refer to the RKM's and initialize it yourself each time, or you can use the built in routine NewList(). Earlier you saw the way to manually initialize a list. It's much easier to initialize a list in the following manner:

```
NewList (&MyList);
```

which will initialize the list "MyList". Note that the argument to NewList() is a struct List * and not a struct List.

Now, let's use some of the list handling routines. Refer to listing 1 for this discussion.

Listing 1 A List Handling Example

```
/****** *
 * June, 1990 Developer's Conference Source Code, *
 * Copyright 1990, Commodore-Amiga *
 * *
```


1

2

3

```

*****
* *
* Program:  LIST.C *
* Compiler:  Lattice 5.04 *
* Synopsis:  example of using Exec's list routines *
* Author:   Kevin-Neil Klop *
* *
*****/
/* Amiga routines include files. */
#include <exec/types.h> /* ALWAYS include this */
#include <exec/nodes.h> /* included for completeness */
#include <exec/lists.h> /* included for obvious reasons */
#include <exec/memory.h> /* Included for memory */
/* allocation routines */
#include <proto/exec.h>
#include <proto/all.h> /* Include the prototypes for */
/* everything. */
/* Lattice routines include files */
#include <stdlib.h> /* For RNG routines */
#include <stdio.h> /* Let's keep I/O simple for */
/* the moment. */
/* Define my node structure, and the parameters for ForgetList */
struct aNode
{
    struct Node aN.Node;
    UWORD aN.Data;
};
void ForgetList(struct List *aList, UWORD TheNodeSize);
/* Define the nodesize constant. */
const UWORD NodeSize = sizeof(struct aNode);
/*****
* Code starts here. *
*****/
main()
{
    UWORD i; /* A counter */
    struct aNode *MyNode; /* A pointer to a node in the */
    struct aNode *Node2; /* list. */
    struct aNode *NextNode;
    struct List MyList; /* Allocate a list structure */
    /* Note that it's a "MinList" */
    /* Meaning "Minimum List". */
    /* None of that extra stuff for */
    /* that's in a full list is */
    /* needed for this example. */
    srand(5); /* Seed the RNG */
    /* Initialize the list itself. We'll do this with one */
    /* of the routines that's in amiga.lib */
    NewList((struct List *)&MyList);
    /*****
    * Allocate a bunch of nodes containing random numbers *
    *****/
    for(i=1; i<40; i++)

```

1

2

3

```

{
    MyNode = (struct aNode *)AllocMem(sizeof(struct aNode),
        MEMF_FAST | MEMF_CLEAR);
    if (!MyNode)
    {
        MyNode = (struct aNode *)AllocMem(sizeof(struct aNode),
            MEMF_CHIP | MEMF_CLEAR);
        if (!MyNode)
        {
            printf("Could not allocate room for node #ForgetList((strn
                List *)&MyList, NodeSize);
            exit(10);
        }
    }
    MyNode->aN_Data = rand();
    AddHead((struct List *)&MyList, (struct Node *)MyNode);
}
/*****
 * Now sort the list *
 *****/
NextNode = (struct aNode *)MyList.lh_Head;
while (NextNode->aN_Node.ln_Succ->ln_Succ != NULL)
{
    MyNode = NextNode;
    NextNode = (struct aNode *)NextNode->aN_Node.ln_Succ;
    Remove((struct Node *)MyNode);
    for (Node2 = (struct aNode *)MyList.lh_Head;
        (Node2->aN_Node.ln_Succ->ln_Succ != NULL) &&
        (MyNode->aN_Data < Node2->aN_Data);
        Node2 = (struct aNode *)Node2->aN_Node.ln_Succ);
    Insert(&MyList, (struct Node *)MyNode, Node2->aN_Node.ln_Pred);
}
/*****
 * And for fun, print it out... *
 *****/
for(MyNode = (struct aNode *) (MyList.lh_Head), i = 1;
    MyNode->aN_Node.ln_Succ->ln_Succ;
    MyNode = (struct aNode *)MyNode->aN_Node.ln_Succ, i++)
    printf("Node #

/* Now deallocate everything. */
ForgetList((struct List *)&MyList, NodeSize);
exit(0);
}
/*****
 * Release all the entries in the list back to the *
 * System (As mommy always told me, Clean up after myself! *
 * *
 *****/
void
ForgetList(struct List *aList, UWORD TheNodeSize)
{
    struct Node *TheNode, *NextNode;
    for(TheNode = aList->lh_Head,

```



```

        NextNode = TheNode->ln_Succ;
        TheNode->ln_Succ->ln_Succ;
        TheNode = NextNode, NextNode = NextNode->ln_Succ)
        FreeMem(TheNode, TheNodeSize);
    NewList((struct List *)aList);
}

```

Skipping all the include files, we get to the point where a struct `aNode` is declared. An `aNode` consists of a Node and some data. Further down, inside the `main()` procedure, a list called `MyList` is declared. `NewList()` is called to initialize the list. Next nodes are allocated from Amiga memory using the Amiga `AllocMem()` call ¹. If, for some reason a node can not be allocated, then the program cleans up after itself by calling its own `ForgetList()` function ², and then exits. Otherwise the node's data field is filled in with a random value and then added to the list.

Once the list has been created, representing some list of data, the list is sorted. `NextNode` is pointed at the head node of the list (the first node that is NOT part of the list header) and then the sort loop is entered. The sort loop will end when `NextNode` is pointing to the last node in the list.

The node that is being sorted into the list is first removed from the list through the `Remove()` call. `Remove()` simply removes a node from a list.

Next the list is searched from the start for the first node whose data \geq to the data that is being sorted into the list. When the proper place for the node is found, it is `Insert()`ed back into the list. This process is repeated until all the nodes are in the proper locations in the list.

After the list is sorted, and all thge nodes are printed, then the list is deallocated in the `ForgetList()` routine. This is an important thing to remember:

Always clean up after yourself!

The Amiga operating system does no "resource tracking" for you. It's all up to you. If you allocate something and never deallocate it, that resource be it memory, a device, or a file, is devoted to your program /sl even if your program is no longer loaded and executing! Well, actually, a reboot will generally free up a resource, but you should not be counting on someone rebooting their computer after your program is done.

The `ForgetList()` call merely starts at the trop of the list and goes through all the nodes in that list, deallocating them as it goes. When it's done, it re-initializes the list header, and then returns to the caller.

9 Ports

One of the main lists that you wil have to deal with goes under the name of a "message port".

In the Amiga, programs communicate with each other and with the operating system mainly through the use of inter-process communications. That's a fancy name for being able to send messages back and forth to each other. In order for your program to take part in this communications, it has to have a message port

¹ To be explained later

² Note that this is defined in the program - it's not a system call

1

2

3

in which to receive messages.

A message port is much like a postal mailbox. Other people send mail (messages) to your mailbox (message port). Once in a while you go out and collect your message, read them, throw out the bills, etc. On the Amiga, other programs and portions of the operating system will be communicating with you through your mail message port in much the same manner. There's nothing all that mysterious about it, really. Mostly, the message port contains a list header on which all your "delivered, but not yet gathered" are kept.

In some ways your message port is like a telephone. It can be listed or unlisted, although we call them "public" and "private". A public message port can be found by anyone - you can get messages from people that you've never heard of through a public message port (sounds somewhat like junk mail, does it not?). On the other hand, if you use a private message port, then you will only get messages from those programs that you've told about it.

Enough about theories, analogies, and junk mail. How does one go about creating a message port?

Well, one of the ways one could go about it is to find out the data structure for a message port, allocate such a structure, fill it in, and then announce it to the world.

Or you can make one function call that will do that all for you.

To create a message port, one calls the `CreatePort` function. `CreatePort` takes two arguments. The first argument is the name of the port. If you put a `NULL` as the first argument, then the port will have no name, and will be made a private port. If you put a pointer to some character string as the first argument, then the port will be named and will also be made a public port.

The second argument specifies the priority of the message port. Only under specialized circumstances should you use a priority that is not zero. Also, note that the priority field is a long value. Thus, if you are specifying a constant, there must be an "l" appended to it.

The inverse of creating a port is deleting it, and the call to delete a port is `DeletePort()`. Note that you ought to remove all messages from the port before you delete the port. Otherwise there will be messages hanging about in limbo with no message port to use in rejoining the rest of the system.

Once you have a message port, and that someone else knows about the port (either because it's public, or because you told someone about it ³ We'll even assume that someone has already sent you a message. How do you get it?

Well, you could start at the top of the list that represents your unreceived mail messages, move down the list to the first message, dequeue it yourself, and then do something with it. Or you can call `GetMsg()`. `GetMsg()` takes one parameter - a pointer to your message port. If there's a message waiting in the port, then a pointer to a struct `Message` is returned to you. In most instances, you will have to cast this return to be a pointer to whatever type of message it is that you're expecting to receive. If there isn't any message waiting for you at that port, then `GetMsg()` returns a `NULL` value.

Each message that you receive that was originated by someone else should be `ReplyMsg()`ed in order to notify the sender that you've seen and acted upon his message - it's now safe for him to reuse or deallocate that message. Note that even though a message is in your mailbox and that you have `GetMsg()`ed it, you do not own that message. The originator of the message continues to own a message.

³ There's many ways to tell someone about your port. We'll get into that later when we talk about Intuition and devices.

1

2

3

Thus, a typical loop to get all your messages is:

```
struct Message *aMessage;
while(!(aMessage = GetMsg(MyPort)));
/* Process the Message in here. */
/* Now assume that the processing of the message is done.*/
ReplyMsg(aMessage);
```

That will allow you to handle all the messages. When all the messages are processed, the loop exits. The problem becomes, what do you do when there are no more messages available to you? In most cases, you want to sit around and wait for some new message to come in. On non-multitasking machines, the accepted solution would be something like this (Note: This is NOT the way to program for the Amiga!!!):

```
while(1)
{
    while(aMessage = GetMsg(MyPort)) {
        /* Process the Message in here. */
        /* Now assume that the processing of the message is done. */
        ReplyMsg(aMessage);
    }
}
```

This is what's known as "busy-waiting", and is severely frowned upon in the Amiga world. What you basically want to do is say, "I have no more messages to process. Operating System, please give CPU time to other people that need it while I wait for another message." The last code fragment did not do that. Instead, it wasted CPU time constantly checking to see if another message had appeared in its message port.

A better way to do this is with something called "signals" on the Amiga.

10 Signals

The basic idea to peaceful co-existence on the the Amiga is that if you're not using something, let someone else use it. This goes for memory, devices, and the CPU itself. If you have to wait for someone else to do something, you should give up the CPU until you're ready to execute again.

But how is this to be accomplished?

The most common thing to be waiting for are message ports. Each message port has something called a "signal bit" associated with it that is set when a message comes into your port. There are also a few system functions that tell the Amiga operating system to stop devoting CPU time to your program until the specified signal bit(s) are set.

You can use these features to either wait for a specific port through the **WaitPort()** call, or use the general purpose **Wait()** function. Let's examine listing 2 for a moment.

Listing 2 Message Receiver

```

/*****
 * June, 1990 Developer's Conference Source Code, *
 * Copyright 1990, Commodore-Amiga *
 * *
 *****/
 *
 * Program:  MESSENGER.C *
 * Compiler: Lattice 5.04 *
 * Synopsis: example of using Exec's list routines *
 * Author:  Kevin-Neil Klop *
 *
 *****/
#include <exec/types.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <stdio.h>
#include <proto/all.h>
#define MessagePortName "MESSAGEE"
main(argc,argv)
int argc;
char *argv[];
{
    struct MsgPort *MyPort;
    struct Message *aMessage;
    MyPort = CreatePort (MessagePortName, 0L);
    if (!MyPort)
    {
        puts("Could not open my own messageport.");
        exit(10);
    };
    while(!(aMessage = GetMsg(MyPort)))
        Wait(1L << MyPort->mp_SigBit);
    do
    {
        puts ("MESSAGEE: I got a message!");
        ReplyMsg(aMessage);
    }
    while(aMessage = GetMsg(MyPort));
    DeletePort (MyPort);
}

```

Listing 2 is nothing more than a program that sits around in memory waiting for a message to appear in its message port. Thus, the first thing that it does is allocate a public message port via the `CreatePort` call. If the port couldn't be created, then the program exits with a simple message. After that, it calls `Wait()` in order to allow other programs the use of the CPU while it has nothing to do. Yes, other programs would get time anyway, but this way Program 2 doesn't use up ANY time while it has nothing to do.

When the program is woken up, most people assume that there's a message in the message port. This is not

necessarily a good assumption. It is better to check and make sure that there's a message waiting. Thus this program is caught in a loop that checks if there's a message, and if there isn't it goes back to waiting. Eventually a message will appear and the loop will be broken.

Now a second loop is entered wherein all messages in the message port are replied to before the message port is closed. Closing the message port first is generally a bad idea. Note that even this strategy may not work as a new message may come in between the time you reply to the last message and the time you delete the port.

Later on we'll learn how to avoid even this possibility.

Now that we've briefly looked at the receiving end of a message handler, let's look at the sending half of the program. Listing 3 contains a program that will send a message to the program in listing 2.

* * *

Listing 3
Message Sender

```
/*
 * June, 1990 Developer's Conference Source Code, *
 * Copyright 1990, Comodore-Amiga *
 * *
 ****
 * *
 * Program:  MESSENGER.C *
 * Compiler:  Lattice 5.04 *
 * Synopsis:  example of using Exec's list routines *
 * Author:   Kevin-Neil Klop *
 * *
 ****/
#include <exec/types.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <stdio.h>
#include <proto/all.h>
#define MessagePortName "MESSAGEE"
main(argc,argv)
int argc;
char *argv[];
{
    struct MsgPort *ThePort;
    struct MsgPort *MyPort;
    struct Message *aMessage;
    MyPort = CreatePort(MessagePortName, 0L);
    if (!MyPort)
    {
        puts("Could not open my own messageport.");
        exit(10);
    }
};
```


FOR BID

PERMIT

```

ThePort = FindPort(MessagePortName);
if (!ThePort)
    puts("Could not find MESSAGEE's port.");
else
{
    aMessage = (struct Message *)AllocMem(sizeof(struct Message),
    MEMF_PUBLIC |
    MEMF_CLEAR);
    if (!aMessage)
        puts("Could not allocate a message to send to MESSAGEE.");
    else
    {
        aMessage->mn_ReplyPort = MyPort;
        PutMsg(ThePort, aMessage);
        puts("MESSENGER:I sent a message, and am waiting for the
        reply...");
        Wait(1L << MyPort->mp_SigBit);
        while(aMessage = GetMsg(MyPort))
        {
            puts("MESSENGER: I received a reply!");
            FreeMem(aMessage, sizeof(struct Message));
        }
    }
}
DeletePort(MyPort);
}

```

Much like listing 2, this program allocates a message port through CreatePort. the difference being that this port is a private port and is NOT placed onto the public message port list buy the CreatePort call.

After the message port is created, this prgram goes out looking for the message port created by listing 2. It does this through the use of the FindPort() call. The FindPort() call taes a name and attempts to find a message port with the same name. Through the use of this mechanism programs can rendezvous with each other (they go looking for each other's port). Alternately, you can tell if someone has already loaded your program by doing a FindPort() on your own message port BEFORE you open it. If you find it, then your program is already in memory and executing.

If Program 3 could not find Program 2's message port, then it outputs a simple message, deletes the port, and exits.

On the other hand, if the port IS found, then a message is allocated, the ReplyPort field of the message initialized to point to Listing 3's port, and then it is sent off to Listing 2's message port. This should (theoretically, at least) wake up Listing 2, which will ReplyMsg() to the message, thus causing it to reappear in Listing 3's "MyPort". It is to receive this reply that Listing 3 needs to create its own message port. If it knew that the message was never to reappear, then it wouldn't need a message port at all.

We have one more important concept to understand before we go on to discussing Intuition.

11 Shared Libraries

Intuition is the name of the part of the Amiga operating system that handles most of the "standard" graphical user interface. It provides calls to open screens, windows, place gadgets in them, and provide events corresponding to user actions. It's something that almost all programs have to deal with if they want to interact with the user.

Intuition is implemented as a shared library in the Amiga operating system. A shared library is a little different than the normal libraries that you might be used to. You do not link to a shared library during compilation/linking. Instead, you open the library during run time. Exec provides a call to do this for you called `OpenLibrary()`.

`OpenLibrary()` returns to you a pointer to the library that is in memory. In this way, many people can use the same library (which is why it's called a "shared library"). Since Intuition is almost 98K in OS 2.0, there would not be enough RAM to run more than 5 programs that use Intuition at the same time in a standard 1-Megabyte machine (419K for the operating system, leaving 580K+).

Using C, you can pretty much treat a shared library as if it had been linked to your program in the same way as other libraries, except that you have to open the library at the beginning of your program and close it at the end of your program.

Note that it is not a good idea to close the library until you are done with. Although such a scheme will work with some libraries, you are running the risk that the library will be purged from memory while you still need it. Libraries that are currently opened by someone (their "in-use count" > 0) are ineligible for purging. By the same token, if you don't need a library, don't open it, as that will sometimes just waste memory.

In order to open a library, you need to declare a global value that will be the "base" pointer for that library. In almost all cases, the name of that variable should be the name of the library suffixed by "Base". Case of the spelling IS significant in C.

Thus, the base pointer for the intuition library would be `IntuitionBase`, the base pointer for exec would be `ExecBase`, the base pointer for graphics would be `GfxBase` (well, it's CLOSE to "graphics"). The type of that variable is a pointer to a structure with the same name. Thus, the complete declaration of `IntuitionBase` would be:

```
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
etc.
```

Remember, these pointers **MUST** be globals!

In order to open the intuition library, call `OpenLibrary()` thusly:

```
IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", 0L);
```

You should check the value of the returned base pointer. If it's `NULL`, then the specified library could not be opened.

The little number on the end of the call (0L in the example) specifies the revision of the library that you require. The `OpenLibrary()` call will only succeed if it can find a copy of the library with a revision number greater than or equal to the revision label you specify. Setting that parameter to 0L means that you will

accept any copy of the library that can be found. In general, you should be using the smallest number here that you possibly can.

For example, if you're using the BOOPSI 4 features of, then you will need to specify 36L to make sure that you have a copy of intuition that can handle BOOPSI calls.

To close the library, you use the `CloseLibrary()` call with the library base pointer as the first argument. Thus:

```
CloseLibrary((struct Library *)IntuitionBase);
```

Notice that the base pointer is cast to the generic type of "Library *"

12 Intuition

Intuition is responsible for handling almost all of the user interface. It is through Intuition that you create screens, windows, menus, gadgets for your windows, etc. It is implemented as a shared library, and thus you will need to open it via the `Exec OpenDevice()` call.

Central to the concept of dealing Intuition is the Intuition Direct Communications Port 5. That's a daunting name for nothing more than a message port, which we discussed earlier.

Intuition notifies you of events that occur by sending messages to the IDCMP. Thus, to wait on something that the user does, you simply wait on the IDCMP for a message to appear.

IDCMPs are associated with windows. Not screens, but windows, and they are created for you when the window is created. This implies that in order to interact with the user, you **MUST** have a window somewhere. If you have a screen and want to avoid the appearance of having a window, that's possible. Nevertheless, Under normal conditions you **MUST** have a window in order to deal with the user.

To create a window, you must allocate and initialize something called a `NewWindow` structure, then call `OpenWindow()` to display the window on the screen. The `OpenWindow()` call will return to you a pointer to a `Window` structure, and it is through the window structure that you will control your new window.

Following is an example of allocating a `NewWindow` structure, filling it out, opening a window, deallocating the `NewWindow` structure, and then closing the window.

* * *

Listing 4 Creating and Destroying a Window

4 See the BOOPSI talk by Jim Mackraz for more information on this

5 This are normally known as an IDCMP

1

2

3

```

#include <exec/types.h>
#include <intuition/intuition.h>
struct IntuitionBase *IntuitionBase;
main()
{
    struct NewWindow *MyNewWindow;
    struct Window *MyWindow;
    if(IntuitionBase = OpenLibrary("intuition.library", 34L))
    {
        if(MyNewWindow = (struct NewWindow *)
            AllocMem(sizeof(struct NewWindow), MEMF_PUBLIC | MEMF_CLEAR)
        {
            MyNewWindow->LeftEdge = MyNewWindow->TopEdge = 0;
            MyNewWindow->Width = 200;
            MyNewWindow->Height = 200;
            MyNewWindow->DetailPen = 0;
            MyNewWindow->BlockPen = 1;
            MyNewWindow->Title = "This Is My Window";
            MyNewWindow->MinWidth = MyNewWindow->MaxWidth = 200;
            MyNewWindow->MinHeight = MyNewWindow->MaxHeight = 200;
            MyNewWindow->Type = WBENCHSCREEN;
            MyWindow = OpenWindow(MyNewWindow);
            CloseWindow(MyWindow);
        } ;
        CloseLibrary((struct Library *)IntuitionBase);
    }
}

```

This will open a window on the workbench screen, then close it. The operation will happen probably a little bit faster than you really want to, but it at least demonstrates the simplest form of opening and closing the window.

Such a window will not be able to handle user operations because none of the IDCMPFlags have been set, and thus Intuition will not be sending you any messages notifying you of user actions. To do that, you need to set the various IDCMPFlags that corresponds to the events that you will be interested in. These flags are explained in the RKMs and in the intuition.h file.

Listing 5 Example of IDCMP Handling

```

/** ----- */
/* window.c - A demo of how to open a window on the Workbench screen */
/* and then handle a CLOSEWINDOW gadget via the Window's */
/* IDCMP. */
/* ----- */
#include <intuition/intuition.h>
#include <exec/types.h>

```

1

2

3

```

VOID Getout( char *inform, SHORT errorval ) ;
struct NewWindow nw = {
    0,0,
    640,200,
    1,2,
    CLOSEWINDOW,
    WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE,
    NULL,
    NULL,
    (UBYTE *)"MY WINDOW EXAMPLE",
    NULL,
    NULL,
    -1,-1,-1,-1,
    WBENCHSCREEN
};
struct Window *MyWindow = NULL ;
struct IntuitionBase *IntuitionBase = NULL ;
struct GfxBase *GfxBase = NULL ;
int main( VOID )
{
    ULONG class ;
    BOOL keepgoing = TRUE ;
    struct IntuiMessage *img ;
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", 0L ) ;
    if(IntuitionBase == NULL)
        Getout("Intuition library open failed", 10) ;
    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 0L ) ;
    if( GfxBase == NULL )
        Getout("Graphics library open failed", 20 ) ;
    MyWindow = (struct Window *)OpenWindow( &nw ) ;
    if( MyWindow == NULL )
        Getout( "Window open failed", 30) ;
    while( keepgoing == TRUE )
    {
        Wait( 1L << MyWindow->UserPort->mp_SigBit ) ;
        img = (struct IntuiMessage *)GetMsg( MyWindow->UserPort ) ;
        if( img != NULL )
        {
            class = img->Class ;
            ReplyMsg((struct Message *)img ) ;
            if( class == CLOSEWINDOW )
                keepgoing = FALSE ;
        }
    }
    Getout( NULL, 0) ;
}
/* ----- */
/* Getout function cleans up everything, prints any messages and exits */
/* ----- */
VOID Getout( char *message, SHORT errorval )
{
    if( message != NULL )

```


1

2

3

```

printf("
if( MyWindow != NULL )
    CloseWindow( MyWindow ) ;
if( GfxBase != NULL )
    CloseLibrary( GfxBase ) ;
if( IntuitionBase != NULL )
    CloseLibrary( IntuitionBase ) ;
exit( errorval ) ;
}

```

Sometimes one wishes to create their own screen for the programs. In order to do that, you must allocate and initialize a NewScreen structure, then call the OpenScreen() function. For example:

Listing 6 Opening A Custom Screen

```

#include <exec/types.h>
#include <intuition.h>
main()
{
    struct Window *MyWindow;
    struct Screen *MyScreen;
    struct NewWindow *MyNewWindow;
    struct NewScreen *MyNewScreen;
    if(IntuitionBase = OpenLibrary("intuition.library", 34L))
    {
        if(MyNewScreen = (struct NewScreen *)
            AllocMem(sizeof(struct NewScreen), MEMF_PUBLIC | MEMF_CLEAR)
        {
            if(MyNewWindow = (struct NewWindow *)
                AllocMem(sizeof(struct NewWindow), MEMF_PUBLIC |
                MEMF_CLEAR))
            {
                MyNewScreen->LeftEdge = 0;
                MyNewScreen->TopEdge = 0;
                MyNewScreen->Width = 640;
                MyNewScreen->Height = 200;
                MyNewScreen->Depth = 2;
                MyNewScreen->DetailPen = 1;
                MyNewScreen->BlockPen = 0;
                MyNewScreen->ViewModes = HIRES;
                MyNewScreen->Type = CUSTOMSCREEN;
                MyNewScreen->Font = NULL;
                MyNewScreen->DefaultTitle = (UBYTE *) "My Screen Title";
                MyNewScreen->Gadget = NULL;
                MyNewScreen->CustomBitMap = NULL;
            }
        }
    }
}

```

1

2

3

```

        if(MyScreen = OpenScreen(&MyNewScreen))
        {
            MyNewWindow->LeftEdge = MyNewWindow->TopEdge = 0;
            MyNewWindow->Width = 200;
            MyNewWindow->Height = 200;
            MyNewWindow->DetailPen = 0;
            MyNewWindow->BlockPen = 1;
            MyNewWindow->Title = (UBYTE *)"This Is My Window";
            MyNewWindow->Screen = MyScreen;
            MyNewWindow->MinWidth = MyNewWindow->MaxWidth = 200;
            MyNewWindow->MinHeight = MyNewWindow->MaxHeight = 200;
            MyNewWindow->Type = CUSTOMSCREEN;
            if(MyWindow = OpenWindow(MyNewWindow))
                CloseWindow(MyWindow);
            CloseScreen(MyScreen);
        }
    }
}
CloseLibrary((struct Library *)IntuitionBase);
}

```

Again, the opening and closing of the screen and window will most likely happen faster than you really want it to.

13 Intuition and Text

There's a structure that is basic to dealing with text in Intuition. This structure is called `IntuiText`.

An `IntuiText` structure defines the pen colors for the block pen and the detail pen, the drawing mode to be used in rendering the text, the relative starting location for the text in the horizontal and vertical directions, the text attributes (font), a pointer to the string to be rendered, and a link to the next `IntuiText` structure.

The layout of the structure is:

```

struct IntuiText
{
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    SHORT LeftEdge, TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
};

```


We'll show more about IntuiText initialization in the next section when we talk about menus.

14 Intuition Menus

Intuition menus, or as they are more properly known "pull down menus" are implemented through three basic structures - IntuiText to represent the textual contents of menus, MenuItem structures that link together the items contained within a single menu, and Menu structures that link together several menus. For example, let us assume that there is a menu bar for the Amiga that looks like:

Project	Edit	Control	Timing
Save	Cut	Start Motor	Delay until ...
Save as	Copy	Stop Motor	Synchronize ...
Recall	Paste	Obtain Sense Data	Reset Timer
	Delete		

All the text in the above example are kept in IntuiText structures, as outlined in the previous section. The four menu headers ("Project", "Edit", "Control", and "Timing") are kept in 4 Menu structures that in turn point to MenuItem structures that contain the other elements.

Normally these groups of things are kept in arrays of structures to make life easier in dealing with them, but this need not be the case.

Once all the structures are created and initialized, then they are attached to a window via the Intuition function SetMenuStrip(). SetMenuStrip() uses the following format:

```
SetMenuStrip(window, menu)
```

where *window* is a pointer to the window to which the menu is to be attached, and the *menu* is a pointer to a menu structure that is to be attached.

Once the menu has been attached to a window, and assuming that you have specified that you want to receive MENU PICK messages from Intuition, you will begin to receive messages in the window's message port every time the user selects a menu. Note that the user may select more than one menu choice per message, so the processing of a MENU PICK message involves a loop.

The first thing that you need to do after determining that the message is a menu pick message is check whether the code field of the IntuiMessage contains a MENUNULL code. If it does, then there is not really a menu item that was selected - Intuition is basically telling you that the user started to pick something from the menus, but then changed his mind and didn't pick anything after all.

If the code is not MENUNULL, then you can get a pointer to the MenuItem structure that defines the menu choice the user selected through the use of the ItemAddress() call. ItemAddress() takes as its arguments a pointer to the menu structure and the code field from the IntuiMessage you received. It returns a pointer to the MenuItem associated with the user's selection.

For example, if I had something akin to:

```
struct MyMenuItem ProjectItem[] =
```



```

{
    { /* first menu item (#0) */
        (struct MenuItem *)&ProjectItem[1], /* ptr to next item */
        0, /* left edge */
        0, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[0], /* ItemFill */
        NULL, /* select fill */
        '1', /* COMMSEQ key = '1' */
        NULL, /* SubItem */
        0, /* NextSelect */
    },
    {
        (struct MenuItem *)&ProjectItem[2], /* next item */
        0, /* left edge */
        13, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[1], /* ItemFill */
        NULL, /* select fill */
        '2', /* COMMSEQ key = '2' */
        NULL, /* SubItem */
        0, /* NextSelect */
    },
    {
        NULL, /* ptr to next item */
        0, /* left edge */
        26, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[2], /* ItemFill */
        NULL, /* select fill */
        '3', /* COMMSEQ = '3' */
        NULL, /* SubItem */
        0, /* NextSelect */
    }
} ;

```

And the user selected the second menu item in the Project menu, then `ItemAddress(MyMenu,theIntuiMessage->code)` would return a pointer to `ProjectItem[1]`.

Other useful things that you can do with the code field is run them through the three macros `MENUNUM()`, `ITEMNUM()`, and `SUBNUM()`. `MENUNUM()` returns the menu number the selection is in. The farthest left menu is number 0, the next one is 1, etc. `ITEMNUM()` returns the item number in the menu. The top item in a menu is item number 0, the second one is item number 1, the third item down is item number 2, etc. `SUBNUM()` is used if there is a sub menu (i.e. one sticking out to the right) of the

1

2

3

menu. In this case, SUBNUM() does the same thing as ITEMNUM() only for the sub menu.

Using these numbers, it is possible to construct a jump table for use in processing menu picks, as the following source code shows:

Listing 7
Menupick Example Code

```
/* =====
* menu selection code example - demonstrates how to take incoming menu
* message from Intuition and process them in an application program.
* Handles Multiple Menu Selections (NextSelect).
* =====
*/
struct IntuiText ProjectText[] =
{
    { 1,0,JAM2,0,1,NULL, (UBYTE *) " MENU ITEM ONE ", NULL} ,
    { 1,0,JAM2,0,1,NULL, (UBYTE *) " MENU ITEM TWO ", NULL} ,
    { 1,0,JAM2,0,1,NULL, (UBYTE *) " MENU ITEM THREE ", NULL}
} ;
struct MenuItem ProjectItem[] =
{
    { /* first menu item (#0) */
        (struct MenuItem *)&ProjectItem[1], /* ptr to next item */
        0, /* left edge */
        0, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[0], /* ItemFill */
        NULL, /* select fill */
        '1', /* COMMSEQ key = '1' */
        NULL, /* SubItem */
        0, /* NextSelect */
    },
    {
        (struct MenuItem *)&ProjectItem[2], /* next item */
        0, /* left edge */
        13, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[1], /* ItemFill */
        NULL, /* select fill */
        '2', /* COMMSEQ key = '2' */
        NULL, /* SubItem */
        0, /* NextSelect */
    }
}
```

—

—

—

```

    NULL, /* ptr to next item */
    0, /* left edge */
    26, /* top edge */
    120, /* width */
    11, /* height */
    ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, /* flags */
    0L, /* Mutual Exclude */
    (APTR)&EditText[2], /* ItemFill */
    NULL, /* select fill */
    '6', /* COMMSEQ = '6' */
    NULL, /* SubItem */
    0, /* NextSelect */
}
};
/* The MENU struct */
struct Menu menu[] =
{
    {
        (struct Menu *)&menu[1], /* next menu */
        0,0, /* left, top */
        88,11, /* wide, high */
        MENUENABLED, /* flags */
        (BYTE *)" PROJECT ", /* name */
        (struct MenuItem *)&ProjectItem[0], /* first item */
    },
    {
        NULL, /* next menu */
        88,11, /* left, top */
        48,11, /* wide, high */
        MENUENABLED, /* flags */
        (BYTE *)" EDIT ", /* name */
        (struct MenuItem *)&EditItem[0], /* first item */
    }
};
/* ===== MAIN ===== */
int main()
{
    ....
    struct IntuiMessage *imsg = NULL ;
    ULONG class ;
    USHORT code ;
    if(!SetMenuStrip(MyWindow, menu))
    {
        puts("Could not create the menu strip. exiting...");
        CleanExit(20);
    }
    while( imsg = (struct IntuiMessage *)GetMsg( UserPort ))
    {
        if( imsg != NULL )
        {
            class = imsg->Class ;
            code = imsg->Code ;
            ReplyMsg( imsg ) ;
            if( class == MENUPICK && code != MENUNULL )

```

1

2

3

```

    },
    {
        NULL, /* ptr to next item */
        0, /* left edge */
        26, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&ProjectText[2], /* ItemFill */
        NULL, /* select fill */
        '3', /* COMMSEQ = '3' */
        NULL, /* SubItem */
        0, /* NextSelect */
    }
};

struct IntuiText EditText[] =
{
    { 1, 0, JAM2, 0, 1, NULL, (UBYTE *) " MENU ITEM FOUR ", NULL },
    { 1, 0, JAM2, 0, 1, NULL, (UBYTE *) " MENU ITEM FIVE ", NULL },
    { 1, 0, JAM2, 0, 1, NULL, (UBYTE *) " MENU ITEM SIX ", NULL }
};

struct MenuItem EditItem[] =
{
    { /* first menu item (#0) */
        (struct MenuItem *)&EditItem[1], /* ptr to next item */
        0, /* left edge */
        0, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&EditText[0], /* ItemFill */
        NULL, /* select fill */
        '4', /* COMMSEQ key = '4' */
        NULL, /* SubItem */
        0, /* NextSelect */
    },
    {
        (struct MenuItem *)&EditItem[2], /* next item */
        0, /* left edge */
        13, /* top edge */
        120, /* width */
        11, /* height */
        ITEMTEXT|COMMSEQ|HIGHCOMP, /* flags */
        0L, /* Mutual Exclude */
        (APTR)&EditText[1], /* ItemFill */
        NULL, /* select fill */
        '5', /* COMMSEQ key = '5' */
        NULL, /* SubItem */
        0, /* NextSelect */
    },
    {

```

1

2

3

```

        {
            HandleMenuPick( code ) ;
        }
    }
}
/* =====
* HANDLEMENUPICK
*
* This function takes an intuition 'Code' value (an Intuition MENUNUM),
* extracts the menu, item and subitem numbers which allow you to
* process any of your menu items.
* =====
*/
VOID HandleMenuPick( USHORT code )
{
    struct MenuItem *item = NULL ;
    USHORT menunum ;
    USHORT itemnum ;
    USHORT subnum ;
    while( code != MENUNULL )
    {
        item = (struct MenuItem *)ItemAddress( &menu, (LONG)code) ;
        if( item )
        {
            menunum = MENUNUM( code) ;
            itemnum = ITEMNUM( code) ;
            subnum = SUBNUM( code) ;
            /* Now you know exactly which menuitem was selected and
            you */
            /* can use values *here* to process the selected menu item(s)
            */
        }
        code = item->.NextSelect ;
    }
    return(0L) ;
}

```

15 Graphics

There are basically a few things that the Amiga is truly known for. Foremost among these are the graphics capabilities. While many programs are written without ever having to get more into the graphics than dealing with Intuition, some people want to render pretty pictures and do animations. For these people, one has to deal with the graphics and layers libraries.

The basic concepts that one needs to deal with Graphics are as follows:

1. A *Bit Plane* is a rectangular section of memory that corresponds to part of an image. If you stacked all

1

2

3

the bit planes together that form a single image, and then drove a spike through one of the bits from the top of the pile to the bottom, the bits skewered on that spike together would make a binary number. This number is used to select the *color register* that will decide the color for the pixel represented by the spike.

2. A *Color Register* is a piece of hardware that holds the Red, Green, and Blue color values to be sent on the screen. Almost all of the color that you see on the screen is through selecting the proper color register to paint it's color into the selected pixel.
3. A *ViewPort* is basically part of an image. It can be thought of as peering through a hole in the fence to view the construction work going on. You can't see the whole construction scene through the hole in the fence, but if you could slide the scene around behind the hole, you could see different parts of it. A ViewPort is like that hole, and the Image can be slid around behind the ViewPort so as to display different parts of the image.

ViewPorts can only be stacked vertically. They can not be placed side by side, and there must be at least one blank line of pixels between them. Thus, you can not have overlapping ViewPorts.

4. A *View* is a set of instructions to the Amiga video hardware. Normally, views are set up for you through using the various graphics calls, although you can construct your own lists and dynamically link them into the system lists, creating various video effects.
5. A *Raster* is a representation of a picture. This picture may be larger than your viewport (as the construction scene can be larger than the part of it you can view through the hole). This Raster is kept in a structure known as a BitMap, which has the maximum size of 1024 by 1024 pixels.

Because the raster can be larger than the viewport, The viewport needs to know what part of the raster to display. This is done through a RasInfo structure. The RasInfo contains a pointer to the bit map of the raster, as well as the variable RyOffset and RxOffset. These two variable specify what pixel should be placed in the top left corner of the ViewPort. If you specify an RyOffset and RxOffset of 0, then the top left corner of the Raster will be displayed in the top left corner of the ViewPort.

Here, in one table, are the relevant structures:

```
struct View
struct ViewPort
struct BitMap
struct RasInfo
```

To initialize a View, you call the **InitView()** routine:

```
struct View MyView;
InitView(&MyView);
```

You would then generally initialize the ViewPort structure like so:

```
struct ViewPort MyViewPort;
InitVPort(&MyViewPort);
MyViewPort.RasInfo = &MyRasInfo;
MyViewPort.DWidth = 40;
MyViewPort.DHeight = 25;
MyViewPort.ColorMap = GetColorMap(4L);
```

—

—

—

```

if(MyViewPort.ColorMap == NULL)
    CleanExit(20);

```

In that example, a viewport 40 pixels wide by 25 pixels high, and 2 bit planes deep has been created. In addition, a Color Map of four entries (2 bit planes deep can specify 4 colors: 00, 01, 10, and 11) is created for the viewport.

After initializing the ViewPort, one needs to initialize the BitMap structure:

```

struct BitMap MyBitMap;
InitBitMap(&MyBitMap, 2, 200, 100);
for(depth = 0; depth < 2; depth++)
{
    MyBitMap.Planes[depth] = (PLANEPTR)AllocRaster(200,100);
    if(MyBitMap.Planes[depth] == NULL)
        CleanExit(20);
}

```

This will allocate a bit map that is 2 bit planes deep, 200 pixels across, by 100 pixels down.

Lastly, the RasInfo structure needs to be initialized:

```

struct RasInfo MyRasInfo;
MyRasInfo.BitMap = &MyBitMap;
MyRasInfo.RxOffset = 0;
MyRasInfo.RyOffset = 0;
MyRasInfo.Next = NULL;

```

There is one thing left to do, and that is to define the colors that we wish to use in this View that we're generating:

```

UWORD colortable[] = {0x000, 0xf00, 0x0f0, 0x00f};
LoadRGB4(&MyViewPort, colortable, 4);

```

This will load the color table for my view port with 4 colors, using the array colortable to initialize the color map values.

We now have all the structures necessary allocated and initialized. Note, though, that we do not yet have a View, we only have the definitions of a ViewPort. To actually create a View, we need to call the **MakeVPort()** call:

```

MakeVPort(&MyView, &MyViewPort);

```

This has created the list of instructions necessary to display the view port. It has not yet, however, connected them with any particular view. This is accomplished through the use of the **MrgCop()** graphics call:

```

MrgCop(&MyView);

```

We now have a complete instruction list to display. One last call to tell the system to start running this list of instructions and we'll actually have something on the screen:


```
LoadView(&MyView);
```

And poof we now have a visible viewport on the screen.

the following program creates and displays a 320 by 200, two bit plane display:

Listing 8 Opening and displaying a ViewPort

```
#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <graphics/view.h>
#include <libraries/dos.h>
#include <proto/all.h>
#include <stdlib.h>
#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200
struct GfxBase *GfxBase;
struct View MyView;
struct ViewPort MyViewPort;
struct BitMap MyBitMap;
UBYTE *displaymem = NULL;
struct View *OldView = NULL;
#define BLACK 0x000
#define RED 0xF00
#define GREEN 0x0F0
#define BLUE 0x00F
VOID
DrawFilledBox(WORD FillColor, WORD Plane)
{
    UBYTE Value;
    WORD BoxHeight, BoxWidth, width;
    BoxWidth = WIDTH / 16;
    BoxHeight = HEIGHT / 2;
    value = ((FillColor & (1 << Plane)) != 0) ? 0xff : 0x00;
    for(; BoxHeight; BoxHeight--)
    {
        for (width = 0; width < BoxWidth; width++)
            *displaymem++ = value;
        displaymem += (BitMap.BytesPerRow - BoxWidth);
    }
}
void freeMemory(VOID)
{

```

1

2

3

```

    WORD depth;
    for (depth = 0; depth < DEPTH; depth++)
    {
        if (MyBitMap.Planes[depth])
            FreeRaster(MyBitMap.Planes[depth], WIDTH, HEIGHT);
    }
    if(MyViewPort.ColorMap)
        FreeColorMap(MyViewPort.ColorMap);
    FreeVPortCopLists(&MyViewPort);
    if (MyView.LOFCprList)
        FreeCprList(MyView.LOFCprList);
}
VOID
CleanExit(int Value)
{
    if(OldView)
    {
        LoadView(OldView);
        WaitTOF();
    }
    FreeMem();
    if(GfxBase)
        CloseLibrary((struct Library *)GfxBase);
    exit(Value);
}
void
main(VOID)
{
    WORD depth, box;
    SHORT BoxOffsets[] = {
        802, 2010, 3218 } ;
    UWORD ColorTable[] = {
        BLACK, RED, GREEN, BLUE } ;
    struct RasInfo MyRasInfo;
    if((GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library", 33L)) == NULL)
        CleanExit(20);
    OldView = GfxBase->ActiveView;
    InitView(&MyView);
    InitVPort(&MyViewPort);
    MyView.ViewPort = &MyViewPort;
    InitBitMap(&MyBitMap, DEPTH, WIDTH, HEIGHT);
    for(depth=0; depth < DEPTH; depth++)
        MyBitMap.Planes[depth] = NULL;
    MyRasInfo.BitMap = &MyBitMap;
    MyRasInfo.RxOffset = 0;
    MyRasInfo.RyOffset = 0;
    MyRasInfo.Next = NULL;
    MyViewPort.RasInfo = &MyRasInfo;
    MyViewPort.DWidth = WIDTH;
    MyViewPort.DHeight = HEIGHT;
    MyViewPort.ColorMap = GetColorMap(4L);
    if(MyViewPort.ColorMap == NULL)

```


1

2

3

```

CleanExit(20);
LoadRGB4(&MyViewPort, ColorTable, 4);
for(depth = 0; depth < DEPTH; depth++)
{
    MyBitMap.Planes[depth] =
        (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
    if(MyBitMap.Planes[depth] == NULL)
        CleanExit(20);
}
MakeVPort(&MyViewPort);
MrgCop(&MyView);
for(depth = 0; depth < DEPTH; depth++)
{
    displaymem = (UBYTE *)MyBitMap.Planes[depth];
    BltClear(displaymem, RASSIZE(WIDTH, HEIGHT), 0);
}
LoadView(&MyView);
for(box = 1; box <= 3; box++)
{
    for(depth = 0; depth < DEPTH; depth++)
    {
        displaymem = MyBitMap.Planes[depth];
        DrawFilledBox(box, depth);
    }
}
Delay(10L * TICKS_PER_SECOND);
CleanExit(0);
}

```

6. A *RastPort* is much like a *RasInfo*, except that it contains even more information than the *RasInfo*. Basically, though, it contains a pointer to a bit map, and extraneous information. If you need to get a rast port, you can open a window, and then get the *RastPort* pointer from it.

16 Device Handling

Lastly comes devices. One uses devices to control many of the Amiga's hardware devices such as the serial port, the parallel port, input ports, and audio output.

Device handling is done through a specialized message passing system based on two main functions: **DoIO()**, **SendIO()**, as well as the ability to stop an IO request with **AbortIO()**, and to check the status of an IO request through the use of **CheckIO()**.

What is this request business, though?

On the Amiga, you very rarely command a device directly. Instead, you ask, or request, that a part of the OS do something for you with the device specified. This is done through sending a special message to the process or task that is handling that piece of hardware. This request message is known as an *IOStdRequest*, although some devices have extensions to an *IOStdRequest* to handle their particular requirements (for

example, the Serial device uses an IOExtSer (IO EXTended SERial request) for its work.

The basic order of operations for dealing with a device is to:

1. Create a message port for dealing with the device.
 2. Allocate and initialize the IO Request.
 3. Open the device.
 4. Control the device through the use of DoIO(), SendIO(), CheckIO(), and AbortIO().
 5. Allow all IO Requests to finish, or abort the outstanding ones.
 6. Close the device.
 7. Free the IO Request.
8. Close the message port.

From earlier sections, you already should know how to create a message port, so we'll skip over that.

The allocation and initialization of the IO request is often device specific, but in generic terms, one used either the CreateExtIO() to create extended IO Requests (including serial device IOExtSer requests) or statically creates them in their program.

Examine the following listing:

Listing 9
Example of Device Handling Code

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/text.h>
#include <devices/timer.h>
#include <devices/narrator.h>
#include <libraries/translator.h>
#include <proto/exec.h>
#include <proto/graphics.h>
#include <proto/intuition.h>
#include <proto/timer.h>
/*=====
* Predefined structures, and global variables follow
*=====*/
struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Library *TranslatorBase = NULL;
```



```

struct NewWindow new.window = {
    0,12, /* left & top edge of window */
    300,100, /* width & height of window */
    0,1, /* detail, and block pen */
    CLOSEWINDOW, /* IDCMP messages to watch for
    - just ONE in this case. */
    WINDOWCLOSE|WINDOWDRAG|SMART_REFRESH|NOCAREREFRESH|ACTIVATE,
    /* Window flags - system gadgets to add,
    activation, and refresh modes. */
    NULL, /* Custom gadget list */
    NULL, /* Custom imagery for window check mark */
    (UBYTE *)"<- Click on me to end demo",
    /* Initial string displayed in windows
    title bar. */
    NULL, /* Custom screen pointer */
    NULL, /* Custom bitmap pointer */
    300,100, /* Minimum width/height */
    300,100, /* Maximum width/height */
    /* Note that since this window is
    not resizable, these values are
    not really important in this
    example.
    */
    WBENCHSCREEN, /* Screen type flag!
    We open this window on the
    Workbench screen.
    */
};

struct Window *window = NULL; /* a pointer to a Window structure */
struct RastPort *rp;
struct timerequest tr;
struct MsgPort *tport = NULL;
struct Message *msg = NULL;
void SetTimer();
void SaySomething();
/*****
* This is where it all begins
*****/
main(argc,argv)
USHORT argc;
char **argv;
{
    LONG error;
    ULONG masks;
    int loop, waiting;
    error=10;
    if(IntuitionBase=(struct IntuitionBase *)
    OpenLibrary("intuition.library",0L))
    {
        error++;
        if(GfxBase=(struct GfxBase *)
        OpenLibrary("graphics.library",0L))
        {

```

1

2

3

```

error++;
if(window=OpenWindow(&new.window))
{
    rp=window->RPort;
    Move(rp, 8L, 30L);
    SetAPen(rp, 3L);
    SetBPen(rp, 2L);
    SetDrMd(rp, JAM2);
    error++;
    if(OpenTimer())
    {
        masks=1L << window->UserPort->mp_SigBit | 1L
        << tport->mp_SigBit;
        loop=TRUE;
        while(loop)
        {
            SetTimer();
            waiting=TRUE;
            Wait(masks);
            /* We only have one message to wait for,
            so we don't have to worry
            about checking for multiple messages in
            this case, nor do we
            have to reply to a timer.device message
            because this
            is just a reply to our original message.
            */
            if(msg=GetMsg(tport))
            {
                SaySomething();
                waiting=FALSE;
            }
            /* if its not a timer.device reply, then
            it must be an
            Intuition message. Chances are there
            will only be
            one message waiting for us. If there
            is more
            in this case its no problem as Intuition
            will any extra messages when closing the
            window. All we are waiting for is a
            CLOSEWINDOW message, so we don't even
            check message
            type here, but we do have to ReplyMsg()
            so
            Intuition knows we are done with the mes-
            sage. If
            we get a CLOSEWINDOW message, and there
            is an
            outstanding timer.device request, we abort
            it.
            */
            else

```



```

        {
            if (msg=GetMsg(window->UserPort))
            {
                ReplyMsg(msg);
                loop=FALSE;
                if (waiting)
                {
                    AbortIO((struct IORe-
                        quest *)&tr);
                    WaitIO((struct IORe-
                        quest *)&tr);
                }
            }
        }
        DeletePort(tport);
        CloseDevice((struct IORequest *)&tr);
    }
    CloseWindow(window);
}
CloseLibrary(GfxBase);
}
CloseLibrary(IntuitionBase);
}

/*****
 * Function to open the timer.device
 *****/
OpenTimer()
{
    LONG open_error;
    open_error=OpenDevice(TIMERNAME,UNIT_MICROHZ,(struct IORequest *)&tr,0L);
    if (open_error == 0L)
    {
        if (tport = CreatePort(0L,0L))
        {
            tr.tr_node.io_Message.mn_Node.ln_Type = NT_MESSAGE;
            tr.tr_node.io_Message.mn_Node.ln_Pri = 0L;
            tr.tr_node.io_Message.mn_Node.ln_Name = NULL;
            return(TRUE);
        }
        CloseDevice((struct IORequest *)&tr);
    }
    return(FALSE);
}

/*****
 * Start timer.device (3 secs)
 *****/
void SetTimer()
{
    tr.tr_node.io_Message.mn_ReplyPort = tport;
    tr.tr_node.io_Command = TR_ADDREQUEST;
    tr.tr_time.tv_secs = 3;
}

```

1

2

3

```

        tr.tr.time.tv_micro = 0;
        SendIO((struct IORequest *)&tr);
    }
    /*****
    * Say something via Narrator.device
    *****/
    void SaySomething(rp)
    struct RastPort *rp;
    {
        struct MsgPort *write_port;
        struct narrator_rb voice_io;
        /* This is the string of text we will say. You can play with this,
        but
        may have to modify size of PhoneBuf to translate it.
        */
        char Englstr[] = "Hi, I am the Amiga!";
        UBYTE PhonBuf[160];
        BYTE audio_chan[] = {
            3,5,10,12 } ;
        LONG error;
        char *err_text;
        err_text="Can't open Translator";
        if(TranslatorBase = (struct Library *)
        OpenLibrary("translator.library",0L))
        {
            err_text="Can't open Narrator ";
            error=OpenDevice("narrator.device",0L,(struct IORequest *)&voice_io,0L
            if(error==0)
            {
                err_text="Can't open write port";
                if(write_port = CreatePort(0L,0L))
                {
                    /* initialize io request message */
                    voice_io.message.ioCommand = CMD_WRITE;
                    voice_io.message.ioOffset = 0L;
                    voice_io.message.ioMessage.mn.Node.ln.Type = NT_MESSAGE;
                    voice_io.message.ioMessage.mn.Length = sizeof(voice_io);
                    voice_io.message.ioMessage.mn.ReplyPort = write_port;
                    /* initialize extended info */
                    voice_io.ch_masks = audio_chan;
                    voice_io.nm_masks = sizeof(audio_chan);
                    voice_io.mouths = 1;
                    /* default speech */
                    voice_io.rate = DEFRATE;
                    voice_io.pitch = DEFPITCH;
                    voice_io.mode = DEFMODE;
                    voice_io.sex = DEFSEX;
                    voice_io.volume = DEFVOL;
                    voice_io.sampfreq = DEFFREQ;
                    /* Use Translator to create phonetic string */
                    err_text="Can't Translate text ";
                    error=Translate((UBYTE *)Englstr,
                    (LONG)strlen(Englstr),

```



```

        (APTR)&PhonBuf[0],
        160L);
        if(error == 0)
        {
            voice_io.message.ioData = (APTR)&PhonBuf[0];
            voice_io.message.ioLength = strlen(&PhonBuf[0]);
            DoIO((struct IORequest *)&voice_io);
            err_text=0x0;
        }
        DeletePort(write_port);
    }
    CloseDevice((struct IORequest *)&voice_io);
}
CloseLibrary(TranslatorBase);
}
if(*err_text) Text(rp,err_text,(LONG)strlen(err_text));
}

```

First, the various libraries are opened - Intuition and graphics. Next a window is opened to display messages and text. The A and B pens are set so that things rendered via graphics calls are rendered with color 3 on a color 2 background, using the JAM1 drawing method.

Next a function is called to open the timer device. The timer uses a standard IORequest that was statically created as a global up near the top. If the device is opened successfully, then the message port is created to receive replies to the messages sent to the timer device, and the timer request is initialized. If the timer is successfully opened, then a mask is set up consisting of the signal bits associated with the IDCMP and the message port used to process timer requests.

A loop is entered that will cause the timer device to reply to the message (i.e. send it back to the port pointed to by "tport") in three seconds. Next, the program executes a Wait() in order to wait for the first message that comes back. The message will either be a CloseWindow message through IDCMP, or a reply to the timer request. If there is a message in tport, then we try and output a little message, send another timer request in, and wait once again. This continues until the user clicks on the close gadget in the window to stop the demo.

17 Closing Comments and Bibliography

While this has been far from a complete course in programming for the Amiga, it is hoped that you have absorbed enough ideas to get you started. There are many good books to help you in your programming endeavours. Some of these are listed below:

1. Amiga Rom Kernel Reference Manual: Includes and Autodocs from Addison Wesley. This contains a complete list of the 1.3.2 include files, system calls, and auto docs. One of the "Must Have"s to do any sort of serious programming for the Amiga.
2. Amiga Rom Kernel Reference Manual: Libraries and Devices from Addison Wesley. This contains discussions and in-depth information about dealing with the various pieces of the operating system. While the RKM:Includes and Autodocs is a dry list of functions and data structures, the RKM:Libraries and Devices

2

2

2

contains more explanatory information about the how to deal with things.

3. The Guru Meditations Guide, Volume I, Sassenrath Research. A technical discussion about Exec and the philosophy that went into writing it by the man most responsible for Exec, Carl Sassenrath.

There are many more useful books to the Amiga programmer. Check your library and with other Amiga programmers for more book listings.

In addition, there are many people willing to help each other out. Try lookin on BiX (the Byte Information Exchange), GENie (General Electric Network for Information Exchange), CompuServ, and your local BBS's for source code examples.

Another good source of code examples is the Fred Fish Freely Redistributable Library. This contains hundreds of disks containing games, utilities, and other programs, many with source.

Lastly, experiment, experiment, experiment. Play by the rules in the RKMs if you want to insure compatibility, or at least minimize incompatibility, with future releases of the operating system, but other than that, try writing small example code for yourself about things that you're not clear on.

—

—

—

This PDF has been kindly provided for scan by Bo Zimmerman (<http://www.zimmers.net/cbmpics>). The book clearly has some missing parts, therefore this PDF is not 100% complete. An extra effort has been made to reproduce the exact book structure.

Hardware/Software used to digitize this tome:

- Fujitsu ScanSnap S1300i
- Adobe Acrobat XI Pro
- GIMP

For any suggestions, contact: jman@storiepvtride.it

-- jman
20140920